

LANSA Integrator Guide

- [About this Guide](#)
- [Introduction to LANSAs Integrator](#)
- [Java Service Manager Administration](#)
- [Java Service Manager Clients](#)
- [Java Service Manager HTTP Extensions](#)
- [Java Service Manager Services](#)
- [Create Your Own Services](#)
- [Integrator Studio](#)
- [Integrator User Agent](#)
- [Remote Function Invocation](#)
- [Troubleshooting](#)
- [Recommendations](#)
- [Tutorials](#)
- [Appendix A. Performance and Tuning](#)
- [Appendix B. Client Application Examples](#)
- [Appendix C. EDI Services](#)
- [Appendix D. LANSAs AS2 and AS3](#)

Edition Date March 18, 2015 EPC132800 UI Update

© LANSAs

About this Guide

Before you begin to use this guide, you may wish to review the following:

[What is LANSA Integrator?](#)

[Who should use this Guide](#)

[How to use this Guide](#)

[Samples and Examples](#)

[Additional Information](#)

[Installation and Licensing](#)

For the latest product information and updates, please refer to the LANSA Product Support at the [LANSA Web site](#).

What is LANSA Integrator?

LANSA Integrator enables the integration of Application-to-Application (A2A) and Business-to-Business (B2B) transactions through XML, JSON and Java services.

Using the shipped Services and the Java Service Manager, you can integrate with any system on any platform using your choice of B2B technologies such as JSON and XML parsing, FTP, FTPS, SFTP, HTTP, HTTPS, SMTP, POP3, SOAP and JMS. LANSA Integrator hides the complexities of interfacing with these technologies, letting you concentrate on the business requirements.

LANSA Integrator is based on a Java Service Framework that allows easy integration between Java programs and LANSA, RPG and COBOL applications.

For further information, refer to [How to use this Guide](#).

Who should use this Guide

This guide has been written for developers who will be using LANSA Integrator, in particular the Java Service Manager, to interact with Java classes (programs).

It is assumed that the reader has some knowledge of the target operating system (for example i5/OS, Windows or Linux). If you are developing your own services or Java classes, you must have a working knowledge of Java.

If you are a LANSA developer and will be using the Java classes shipped with the Java Service Manager, detailed knowledge of Java is not required. RDML programming knowledge is a prerequisite.

A fundamental understanding of Java and the Java virtual machine environment is an asset in understanding the overall architecture and operation of the Java Service Manager.

For information about installing LANSA Integrator, refer to the *Installing LANSA on IBM i* Guide, the *Installing LANSA on Windows* Guide or the *Installing LANSA on Linux* Guide. If you are installing LANSA Integrator with no other installation site in your vicinity, please refer to the *Installing LANSA Integrator* Guide.

How to use this Guide

If you are not familiar with LANSAs Integrator and the Java Service Manager, begin by reviewing the [Introduction to LANSAs Integrator](#).

To learn how to start and configure the Java Service Manager, review the [Java Service Manager Administration](#).

If you are executing the Java Service Manager using the shipped services, review the [Java Service Manager Clients](#) and the [Java Service Manager Services](#).

To see the Services shipped with LANSAs go to [Java Service Manager Services](#). You can, of course, write your own services and how to do so is described in [Create Your Own Services](#).

If you are designing Java classes or tuning an existing application, review [Performance and Tuning](#).

If you encounter any problems during the installation, refer to [Troubleshooting](#).

This guide contains a mixture of detailed technical/user information for Java developers who are writing their own services, and LANSAs developers who are using the shipped services. If you are a LANSAs developer, you will not require the more technical Java details on interfaces, classes and so on.

Samples and Examples

[Appendix B](#) contains a range of JSM client application examples for many of the supplied services.

With the LANSAs documentation, LANSAs ships a formal set of "Samples Examples and Templates" (SET) materials, which cover the complete LANSAs product family, including LANSAs Imports and provides excellent tips and techniques.

For other tips on using the features provided by LANSAs Integrator, refer to the [LANSAs Web site](#).

Additional Information

For more details about LANSA, refer to these guides:

- Installing LANSA on IBM i
- Installing LANSA on Windows
- Installing LANSA on Linux Guide
- LANSA for iSeries User Guide
- LANSA Technical Reference Guide

For information about Java and the IBM i, the following IBM Redbooks may be of assistance:

[SG24-7353 IBM Technology for Java Virtual Machine in IBM i5/OS](#).

This product includes software developed by the Apache Software Foundation. Browse at [The Apache Software Foundation](#).

For samples and examples on how you can use the LANSA Integrator, review the [SET](#) Materials.

For the latest product information, refer to the LANSA product Web site at www.lansa.com/support.

Your feedback will help us improve the overall quality of the LANSA documentation and training. Please email your comments to lansatraining@lansa.com.au

Installation and Licensing

For information about installing LANSA Integrator, refer to the [Installing LANSA on IBM i Guide](#) or [Installing LANSA on Windows Guide](#).

LANSA Integrator License

For Licensing details such as how to apply for a license and how to record it on your machine, refer to [Product Licensing](#) on the LANSA Web site.

Tutorials

Tutorials have been provided for you to become familiar with using LANSA. You may also attend class room training. Ask your local LANSA distributor for course schedules or refer to the [LANSA Web Site's education page](#) for the Integrator class room training dates.

For the latest version of the documentation, please check the LANSA Web Site

What's New in this Version?

- New PGPFFileService.
- New RDMLX BIFs JSMX_BEGIN and JSMX_END.
- ZipService performance improvements.
- ExcelService can now read both XLS and XLSX files.
- SOAP Server Types can now be edited within the SOAP Server Wizard.
- Ad-hoc Server Type parameters can now be created within the SOAP Server Wizard.
- SOAP Agent Wizard can now handle input, output and input/output parameters.
- JSMDirect and JSMProxy can now take name-value pairs on the URL.
- JSMDirect and JSMProxy on IBMi Apache server can now receive chunked transfer encoded content.

1. Introduction to LANSA Integrator

This section introduces the Java Service Manager, a part of the LANSA Integrator. It describes the Java Service Manager technology and its benefits.

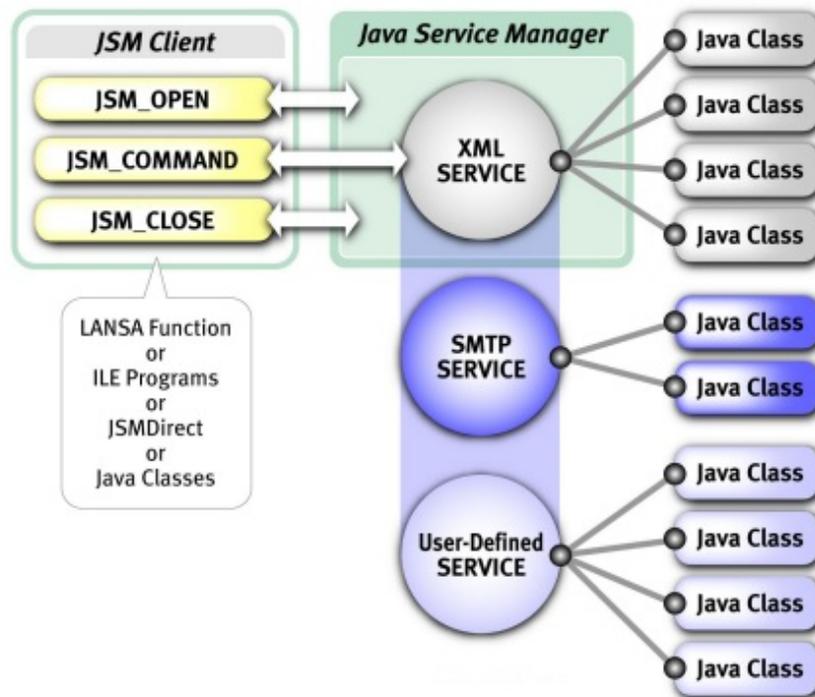
Review the following topics:

[1.1 What is Java Service Manager \(JSM\)?](#)

[1.2 What is Integrator Studio?](#)

[1.3 What is Integrator User Agent?](#)

1.1 What is Java Service Manager (JSM)?



The Java Service Manager:

- Allows integration of Java programs with existing LANSAs or 3GL applications.
- Provides a standardized interface for calling Java classes using Java service programs. When a service is loaded into the JSM, it manages the interface to the services, handles the threading and provides tracing.
- Greatly simplifies the development of server applications requiring the integration of Java Services with LANSAs or 3GL applications.

The Java Service Manager can be used by:

- LANSAs functions (by using LANSAs Built-In Functions (BIFs)).
- LANSAs functions being executed over the Internet using HTTP.
- ILE programs (by calling the Java Service Manager APIs - similar to BIFs).
- Other Java classes.

Some of the services shipped with the Java Service Manager are:

- XMLBindFileService (Local XML file handling).
- XMLBindQueueService (MQSeries XML message handling).

- HTTPService (Generic HTTP service with dynamic content handlers).
- FTPService.
- POP3MailService.
- SMTPMailService.

The JSM includes Java classes for standard requirements such as exception handling, message handling, data conversion, etc.

To understand more about the Java Service Manager, it is recommended that you review:

[1.1.1 Why use Java Service Manager?](#)

[1.1.2 What is a JSM Client?](#)

[1.1.3 What is JSMDirect?](#)

[1.1.4 How does the Java Service Manager work?](#)

[1.1.5 How Can I Use Java Service Manager](#)

[1.1.6 Can I Create My Own Services?](#)

1.1.1 Why use Java Service Manager?

Many vendors and open source communities are implementing their technology using Java classes.

For example, when a customer order is created, an application may want to send a customer an email with the order details as an attachment. This type of application can be achieved by using Java Mail classes with your LANSAs order entry functions. Without the Java Service Manager, this type of application can present a significant technological challenge for traditional business systems developers.

How do you natively call a Java program from your LANSAs or 3GL application?

If you have thirty different Java classes, how will you handle the coding for each class?

How will you manage error handling?

How will you manage the threads and resources for the application?

Manually creating and coding the interface to Java classes can be a challenging and time consuming task that is focused on technology and not your ultimate business solution. The [JSM](#) simplifies the integration of Java classes with your application by standardizing the interface and by managing the execution of the services. It includes a number of Java classes for exception handling, message handling, data conversion, etc. It also provides a comprehensive set of tracing utilities for the application developer. These features can dramatically reduce the time to integrate new Java classes with an application because the developer is not creating these utilities on their own.

- If you are a LANSAs developer, you will call three Built-In Functions.
- If you are an ILE RPG developer you will call four APIs.
- If you are an experienced Java programmer, you can also write your own services to call your own Java classes using the JSM.

The JSM allows you to focus on the business objectives of the applications rather than the underlying technological implementation. By using the Java Service Manager, a complex task or protocol can be simplified to a series of commands. LANSAs developers only need to understand how to use the commands offered by the Java service.

1.1.2 What is a JSM Client?

JSM Client is a program that requests the Java Service Manager to load and run a Java service class.

The client program can be a LANSAs function or an ILE program.

The Java Service Manager supports a tiered architecture and can run on the same or a separate machine to the JSM Client.

Refer to [Java Service Manager Clients](#) for information.

1.1.3 What is JSMDirect?

JSMDirect is a HTTP extension to the Java Service Manager.

It enables a LANSAs or 3GL application using the Java Service Manager to participate in Web services for [B2B](#) transactions.

LANSAs or 3GL applications are executed from a single entry point CGI program.

The JSMDirect CGI program handles the reading of the HTTP request and sending of the HTTP response.

For example, a browser might request the following:

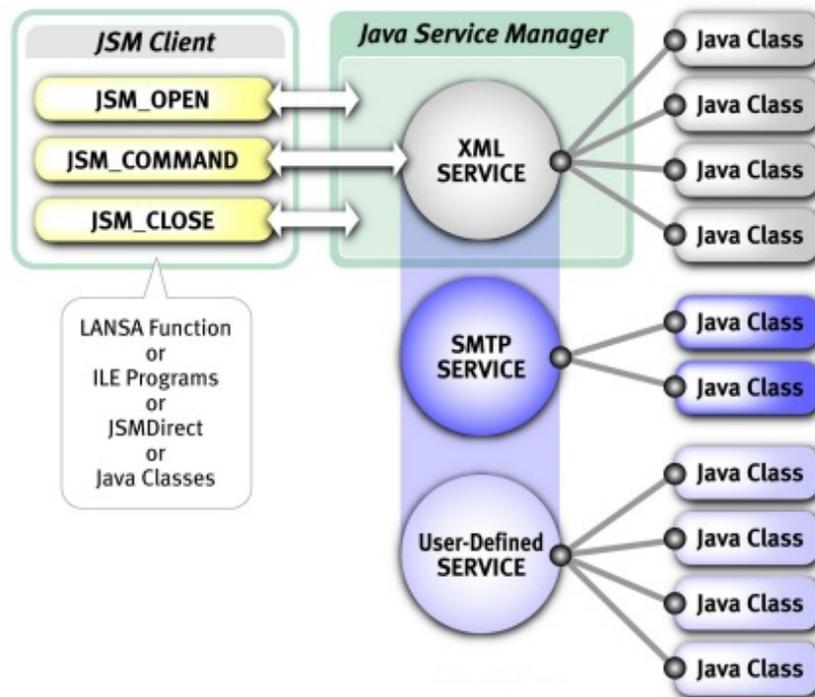
`http://mycompany/cgi-bin/jsmdirect?orderentry`

JSMDirect makes the data transfer to the [JSM](#) transparent to the application developer and greatly simplifies application development. For example, if you are creating a 2-way XML application, your customers will send you an XML document. Using JSMDirect, the XML document can be sent by the customer and a LANSAs function will be run to receive, process the XML document and send a response.

The Java Service Manager does not require the use of LANSAs for the Web. XML parsing and translation are supported directly from the Java Service Manager.

For details on running LANSAs applications with JSMDirect, refer to [Java Service Manager HTTP Extensions](#).

1.1.4 How does the Java Service Manager work?



On IBM i, the Java Service Manager is started using the STRJSM command on the JSM Menu which is described in [Java Service Manager Administration for IBM i](#). The STRJSM command starts a JSM instance in the JSM subsystem. More than one JSM instance can be running. Each JSM instance has its own Java Virtual Machine. The Java Service Manager runs in the JSM subsystem waiting for service requests from a JSM Client.

On Windows, if the JSM Administrator service's Start-up Type is set to Automatic, Microsoft Service Control Manager will start the service automatically during system startup. If it is set to Manual, you need to use Microsoft Service Control Manager to start the JSM Administrator. JSM Administrator starts the Java Virtual Machine and launches the Java Service Manager. Once started, the JSM Administrator will wait for service requests from a JSM Client.

On Linux, the JSM Administrator is started with the strjasm program, which starts the Java Virtual Machine and launches the Java Service Manager. The Java Service Manager will then wait for service requests from a JSM Client.

The JSM Client begins by sending a request to open a connection to the Java Service Manager. If the JSM Client is a LANSA function, then the JSM_OPEN

Built-In Function is used to open the connection. The Java Service Manager will start a new thread on the server. Each JSM Client has its own thread managed by the Java Service Manager.

The JSM Client can be a LANSAs function or 3GL program. Using JSMDirect, the LANSAs function can be invoked from a CGI program using an HTTP request from a client. JSMDirect allows the functions to execute over the Internet to support applications such as 2-way XML. The LANSAs function to be executed is defined in the DC@W29 file on IBM i or in file dc_w29.txt on Windows or Linux. JSMDirect automatically handles the reading and writing of the data stream. (Refer to [1.1.3 What is JSMDirect?](#))

Once a thread is started, the JSM Client can issue a series of COMMANDS to load or unload specific service classes. Only one service class can be loaded at a time. The JSM_COMMAND BIF is also used to execute specific commands in the loaded service. For example, once the FTPService has been loaded, commands are used to LOGIN to the FTP server and GET or PUT files. The Java Service Manager interacts with the Java classes used by the service in order to perform the required FTP operations.

Once the JSM Client has finished, it will send a request to close the connection. The JSM_CLOSE BIF is used to close the connection.

1.1.5 How Can I Use Java Service Manager

Application developers can use the Java Service Manager wherever there is a need to integrate Java services with LANSAs functions or other 3GL applications.

Following are examples of how the Java Service Manager can be used:

- You want to write a 2-way XML application to implement A2A or B2B solutions over the Internet. Using the HTTP Service and the XML Wizard, you can create both client and server applications to send and receive the XML documents.
- Business partners want a simple method to send transaction data from PC files to a server application for posting. Using the User Agent, files can be uploaded and processed by the JSM.
- You need to write an application to process XML documents. The XML Wizard and JSM provide XML services to easily write programs to process.
- You have a set of specialized APIs written in Java and you would like to use these with your LANSAs or 3GL applications. Java Services can be written for these classes so that they can be easily integrated into your applications.

1.1.6 Can I Create My Own Services?

The Java Service Manager is extremely flexible. It can be used to implement your own Java classes on the server or if you wish to use third party Java classes, you can create a Java service for these classes. By creating a service you receive all of the benefits of the Java Service Manager.

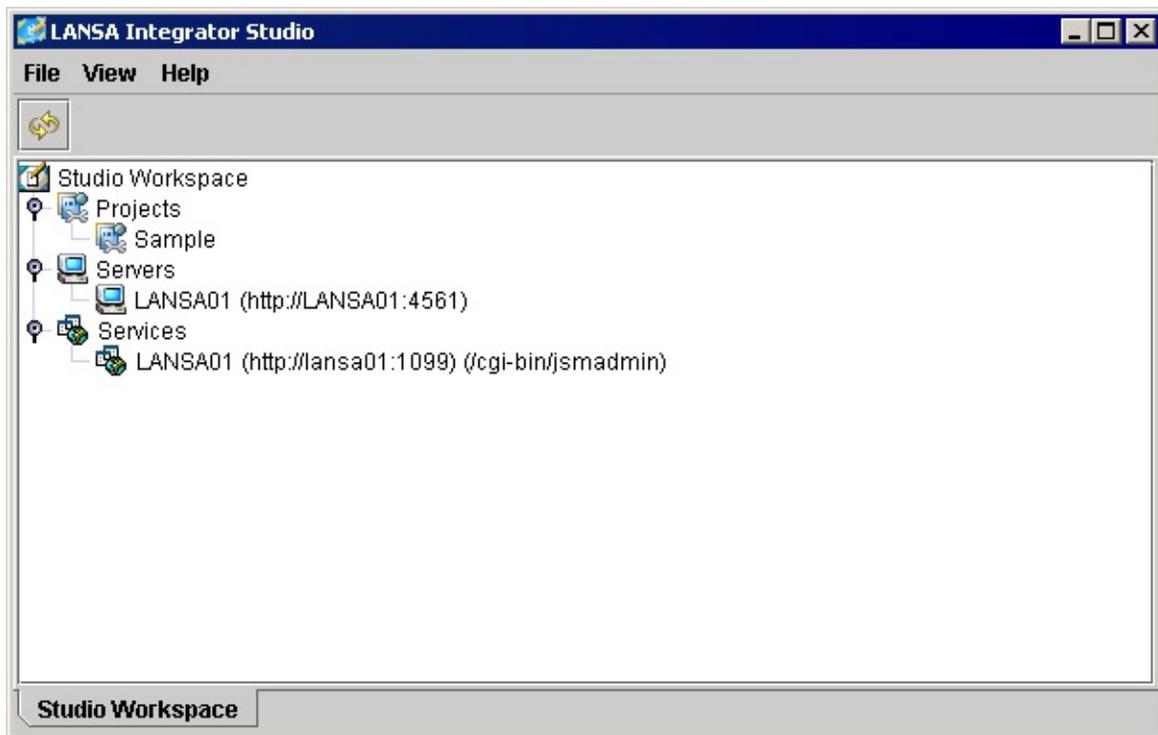
In order to create your own Java service, you must have a working knowledge of Java and you must learn the Java Service Manager interfaces. Refer to [Create Your Own Services](#)

1.2 What is Integrator Studio?

Integrator Studio is an integrated desktop application that allows a single point of management for JSM Server instances as well as the resources used by your Integrator applications or projects. Studio's easy to use graphical interface greatly simplifies the configuration and setup of the JSM Server environment. For example, you can display instance information, update JSM configuration files on the server, download trace files, and publish the generated XSL files to the JSM Server.

Studio allows local management of content using project folders. Project folders allow you to group any files that are part of a specific Integrator application. Projects also allow you to uniquely tag configuration settings within a JSM instance in order to simplify setup tasks. Using Studio, files can be published to or retrieved from the instance server by simple drag and drop operations or using pop-up menus.

Integrator Studio also provides a single point of access for the Integrator tools including the XML Wizard, XML Editor, XSL Compiler and the Soap Wizard. Studio also manages the files created by these tools in order to simplify the management with the JSM Server.



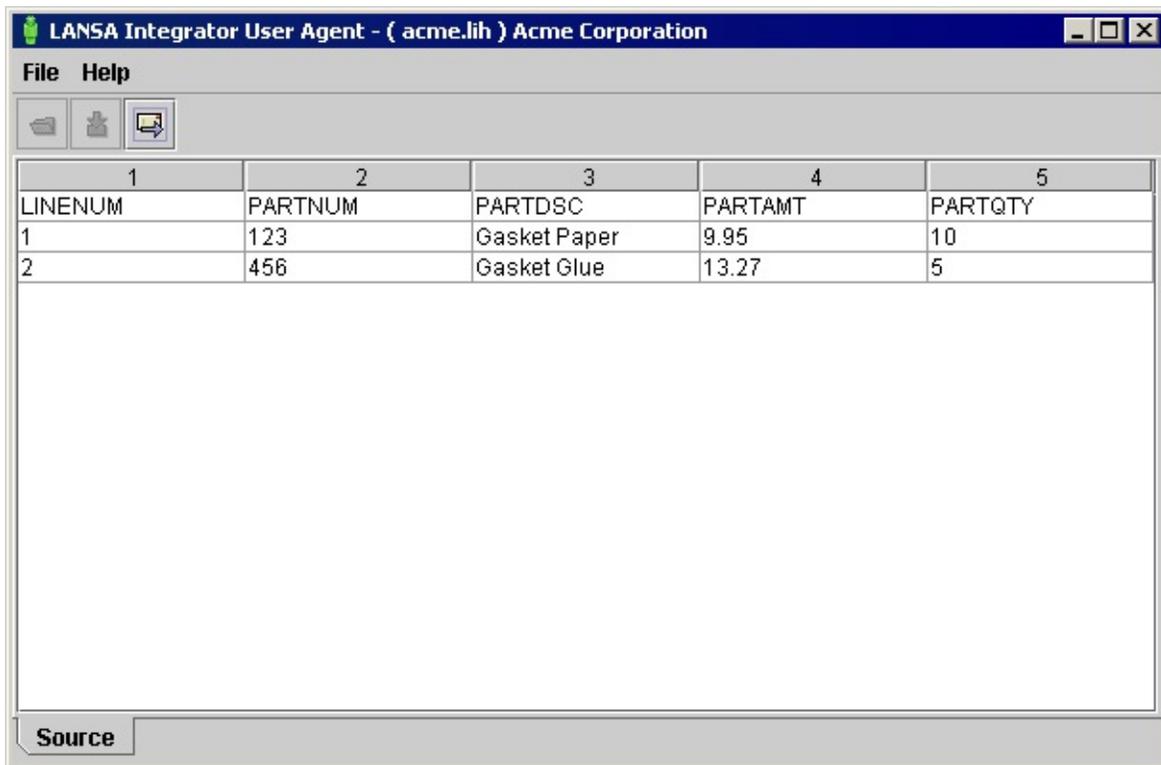
For more information, refer to [Integrator Studio](#).

1.3 What is Integrator User Agent?

The User Agent is an end user (client) application for uploading files to a LANSAs Integrator application. It allows data in Excel spreadsheets, text files (using comma or tab delimited formats) or XML files to be validated and sent from Windows and Linux clients to remote hosts via HTTP, HTTPS, SFTP, FTP or JMS message.

For example, a buying trading partner could enter ordering information in a spreadsheet and send the file to a remote trading hub. The remote trading hub can use a single LANSAs Integrator program to process the files, regardless of format, and return the processed messages back to the user agent.

The User Agent can be run interactively or via a batch process, making it ideal for uploads from workstations to a host for regular tasks such as end of day processing.



1	2	3	4	5
LINENUM	PARTNUM	PARTDSC	PARTAMT	PARTQTY
1	123	Gasket Paper	9.95	10
2	456	Gasket Glue	13.27	5

For more information, refer to [Integrator User Agent](#).

2. Java Service Manager Administration

The administration tasks for the Java Service Manager are specific to the platform on which the JSM Server is installed.

Review the appropriate section:

- [2.1 Java Service Manager Administration for IBM i](#)
- [2.2 Java Service Manager Administration for Windows](#)
- [2.3 Java Service Manager Administration for Linux](#)
- [2.4 Java Service Manager Console](#)
- [2.5 Java Service Manager Refresh](#)
- [2.6 Java Service Manager Pool Server](#)
- [2.7 Java Service Manager Additional Servers](#)
- [2.8 Java Hotspot Technology](#)
- [2.9 Java Endorsed Standards Override Mechanism](#)
- [2.10 JSM Startup Class](#)
- [2.11 Activation Framework](#)
- [2.12 IBM Java System Properties](#)
- [2.13 IBM Technology for Java Shared Classes](#)
- [2.14 LOG4J Logging Services](#)
- [2.15 Axis Properties](#)
- [2.16 Axis Message Handler](#)
- [2.17 SOAP Agent Message Handler](#)
- [2.18 Apache Axis 1.4 WS-Security and WS-Addressing](#)
- [2.19 REST Representational State Transfer](#)
- [2.20 JCE Unlimited Strength Policy Files](#)

2.1 Java Service Manager Administration for IBM i

During installation the xxxJSMLIB and QOTHPRDOWN user profiles are created with a default password of LANSA. You need to change these passwords to make your system secure.

[2.1.1 File and Folder Security](#)

[2.1.2 Network Security](#)

[2.1.3 JSM Job Management](#)

[2.1.4 Java Service Administration](#)

2.1.1 File and Folder Security

During installation the xxxJSMLIB and QOTHPRDOWN user profiles are created with a default password of LANSA. You need to change these passwords to make your system secure.

Files and folders in the JSM instance are shipped with the owner being QOTHPRDOWN and *PUBLIC authority of *EXCLUDE. The JSM user specified at install time is granted all data and object authority.

To allow other user profiles to access JSM files and folders you can use the i5/OS user profile primary group or supplemental group feature.

```
CHGUSRPRF USRPRF(MYUSER) GRPPRF(XXXXXXXXXX) OWNER(*U
```

To change the data and object authorities of existing files and folders you can use the CHGJSMAUT, CHGJSMPGP, CHGAUT or CHGPGP commands.

By default, IFS files created by Java, inherit the *PUBLIC object authority from the parent directory. The system properties os400.file.create.auth and os400.dir.create.auth can be used to control the *PUBLIC object authority for created files and directories.

Specifying the properties without any values or with unsupported values results in a public authority of *NONE.

```
os400.dir.create.auth=none  
os400.file.create.auth=none
```

To change the *PUBLIC file and folder creation data authority to a particular value requires changing the following properties in the SystemDefault properties file.

```
os400.dir.create.auth=RWX  
os400.file.create.auth=RW
```

2.1.2 Network Security

During installation the xxxJSMLIB and QOTHPRDOWN user profiles are created with a default password of LANSA. You need to change these passwords to make your system secure.

You can restrict access to your JSM instance by using TCP/IP client address filtering.

The JSM instance can be configured to only accept connections from specified TCP/IP clients.

For example, if you are running a JSM instance on your IBM i and the LANSA or RPG client programs are running on the same machine (partition), then you can use the LOOPBACK (127.0.0.1) address.

The JSM server will listen on port 4560 and address 127.0.0.1, and only accepts clients from 127.0.0.1.

Using the LOOPBACK address, means no communication traffic extends to the physical card.

It is impossible for another machine or network scanner to access the TCP/IP interface.

```
tcp.port=4560
tcp.backlog=20
tcp.interface=127.0.0.1
tcp.client.address=127.0.0.1
```

Multihomed LOOPBACK address

You can use multiple LOOPBACK addresses for multiple JSM instances and use the same port number.

```
tcp.port=4560
tcp.interface=127.0.0.1
```

```
tcp.port=4560
tcp.interface=127.0.0.2
```

```
ADDTCPIFC INTNETADR('127.0.0.2') LIND(*LOOPBACK) SUBNETMAS
```

GO CFGTCP

1. Work with TCP/IP interfaces

```
10.2.0.173    255.255.0.0    ETHLINE    *ELAN
127.0.0.1    255.0.0.0     *LOOPBACK  *NONE
127.0.0.2    255.0.0.0     *LOOPBACK  *NONE
```

Remember you need to start the 127.0.0.2 interface.

```
PING '127.0.0.1'
PING '127.0.0.2'
```

```
ADDTCPIFC *LOOPBACK help
```

The interface being changed is the loopback or LOCALHOST interface.
Because processing associated with loopback does not extend to a physical line
This special value must be used for any INTNETADR that has a first octet value

2.1.3 JSM Job Management

During installation the xxxJSMLIB and QOTHPRDOWN user profiles are created with a default password of LANSA. You need to change these passwords to make your system secure.

During the installation process the following i5/OS job management objects are created using the 3 letter install prefix.

xxxJSMLIB	User Profile. JSM Runtime User.
QOTHPRDOWN	User Profile. JSM Object Owner.
xxxJSMLIB	Library.
xxxJSMLIB	Subsystem.
xxxJSMJOB	Job Description. STRJSM Job Description.
xxxJSMJOBA	Job Description. Subsystem Auto-start Job Description.
xxxJSMJOBQ	Job Queue.
xxxJSMOUTQ	Output Queue.
xxxJSMCLS	Class.
/LANSA_xxxjsmlib	IFS Folder.

The STRJSM command will do the following processing steps.

Retrieve the JOB from DCXLOADA04 data area.

RTVDTAARA DTAARA(DCXLOADA04 (421 10)) RTNVAR(&JSMJOB)

Submit RUNJSM using the JOB.

SBMJOB CMD(RUNJSM INSTANCE(&INSTANCE) +
VERSION(&VERSION) OPTION(&OPTION) +
GCHINL(&GCHINL) GCHMAX(&GCHMAX) +
JOB(&JSMJOB)) JOB(&JSMJOB) +
JOB(*LIBL/&JSMJOB) JOBQ(*JOB) +
PRTDEV(*JOB) OUTQ(*JOB) USER(*JOB) +
SYSLIB(*SYSVAL) CURLIB(*CRTDFT) +
INLLIBL(*JOB) LOG(4 00 *SECLVL) +
DSPSBMJOB(*NO) MSGQ(*NONE) CPYENVVAR(*NO)

The configuration of these objects can be viewed using the WRKJOB and WRKSBSD commands.

WRKJOB JOB(xxxJSMLIB/xxxJSMJOB)

User profile : xxxPGMLIB

Job queue : xxxJSMJOBQ

Library : xxxJSMLIB

Output queue : xxxJSMOUTQ

Library : xxxJSMLIB

WRKJOB JOB(xxxJSMLIB/xxxJSMJOBA)

User profile : xxxPGMLIB

Job queue : xxxJSMJOBQ

Library : xxxJSMLIB

Output queue : xxxJSMOUTQ

Library : xxxJSMLIB

Request data: RUNJSM INSTANCE(*DEFAULT) ^

WRKSBSD SBSD(XXXJSMLIB/XXXJSMLIB)

Display Autostart Job Entries

JSMJOB XXXJSMJOBDA XXXJSMLIB

Display Job Queue Entries

10 XXXJSMJOBQ XXXJSMLIB 20

2.1.4 Java Service Administration

During installation the xxxJSMLIB and QOTHPRDOWN user profiles are created with a default password of LANSA. You need to change these passwords to make your system secure.

Use the xxxJSMLIB user profile to administer the JSM instance. The initial library list is set and the JSM Menu will appear at sign on.

JSM JSM Menu

Select one of the following:

1. Start Java Service Manager
2. Clear Java Service Manager
3. Optimize Java Service Manager
4. Edit Manager Properties
5. Edit Service Properties
6. Change Default JSM Instance
7. Work with Java Service Manager

Selection or command

===>STRJSM

F3=Exit F4=Prompt F9=Retrieve F12=Cancel

From this menu, you can perform the following options:

- 2.1.4.1 Start Java Service Manager (STRJSM)
- 2.1.4.2 Clear Java Service Manager (CLRJSM)
- 2.1.4.3 Optimize Java Service Manager (OPTJSM)
- 2.1.4.4 Edit Manager Properties (EDTJSMGR)
- 2.1.4.5 Edit Service Properties (EDTJSMSRV)

2.1.4.6 Change Default Instance (CHGJSMDFT)

2.1.4.7 Work with Java Service Manager (WRKJSM) that opens the 2.1.4.8 Work with Object Links window.

The i5/OS commands are in brackets. For example, you may use option 1 or you may use the i5/OS command STRJSM to start the JSM.

2.1.4.1 Start Java Service Manager (STRJSM)

The JSM subsystem needs to be started using the STRSBS SBSD (JSM) command.

The STRJSM command submits a JSM instance to the JSM subsystem.

The source code for STRJSM is in the QCLSRC source file in the JSM library.

Each JSM instance has its own Java Virtual Machine (JVM).

The JSM instance JVM's current working directory is set to the instance path.

A CLASSPATH is created using the instance classes subdirectory and all the *.jar and *.zip files located in the jar subdirectory.

You can start multiple Java Service Managers with a different instance directory, different interface and port, etc.

When the JSM instance starts, a check is made for the existence of the following sub-directories below the instance directory:

- classes
- jar
- properties
- system
- temp
- trace

The Java Service Manager can run on a separate machine to the JSM Client.

When you select the option to start the Java Service Manager, you will be prompted to enter the following:

INSTANCE

The instance defaults to a value of *DEFAULT. This is the recommended value. The default instance is defined in the JSMMGRDTA data area. This value can be changed using [2.1.4.6 Change Default Instance \(CHGJSMDFT\)](#).

VERSION

The JVM Version defaults to a value of *DEFAULT. This is the recommended value. Valid values are based on the currently supported versions of the JVM.

OPTION

The option defaults to a value of *NONE. Valid values include *VERBOSE, *VERBOSEGC, *NOCLASSGC.

For more information about the OPTION keyword refer to the i5/OS JAVA (Run Java Program) command.

GCHINL

Garbage collect initial size.

The possible value range is 256-240000000 kilobytes.

The default is 131,072 kilobytes or 128M.

For more information about the GCHINL keyword refer to the i5/OS JAVA (Run Java Program) command.

For the IBM Technology for Java the GCHINL keyword value becomes the –Xms value.

GCHMAX

Garbage collect maximum size.

The possible value range is 256-240000000 kilobytes.

The GCHMAX value determines the maximum heap size of the JVM, the default value is *DFT.

The GCHMAX *DFT value for the IBM Classic JVM is *NOMAX.

The GCHMAX *DFT value for the IBM Technology for Java is 2G.

For the IBM Technology for Java the GCHMAX keyword value becomes the –Xmx value.

This default value of 2G is too small for a 64-bit Java environment and needs to be increased.

Do not use the maximum value of 240000000 as the IBM Technology for Java command treats this as a *NOMAX value and defaults back to the 2G value

IBM Technology for Java allocates memory in 256M segments. The –Xmx value should be a multiple of 256M so allocated memory is not wasted.

The 32-bit JDK can only have a process address space of 4096M (16 segments) and some of these segments are reserved.

Segment Count	Segment MB	GCHMAX Value
----------------------	-------------------	---------------------

1	256	262144
2	512	524288
3	768	786432
4	1024	1048576

5	1280	1310720
6	1536	1572864
7	1792	1835008
8	2048	2097152
9	2304	2359296
10	2560	2621440
...		
11	2816	2883584
12	3072	3145728
13	3328	3407872
...		
915	234240	239861760

From experimentation, these are the maximum GCHMAX values for IBM Java Technology for Java JDK's.

32bit JVM's are limited to 3407872 kilobytes:

`STRJSM VERSION(*JVM5032) GCHMAX(3407872)`

64bit JVM's are limited to 239861760 kilobytes:

`STRJSM VERSION(*JVM5064) GCHMAX(239861760)`

If you are using the shared classes option (-Xshareclasses), then the maximum GCHMAX value will be less.

Refer to the following IBM Redbook:

[IBM Technology for Java Virtual Machine in IBM i5/OS.](#)

For more information about the GCHMAX keyword refer to the i5/OS JAVA (Run Java Program) command.

JOB

Name of submitted instance job.

COUNT

The number of times the instance is submitted.

Refer to [2.7 Java Service Manager Additional Servers](#).

Exit Program (IBM i)

The STRJSM command submits the RUNJSM program.

The RUNJSM program will call the CL program RUNJSMEXT if it is found in the library list.

The RUNJSM program calls the CL program RUNJSMEXT when the following life cycle events occur:

- ENTRY Before the Java environment starts.
- REBOOT When the Java environment reboots.
Studio Reboot.
- EXIT When the Java environment ends.
Studio Shutdown or ending of the QJVACMDSRV job.

The source code for this exit program is stored in QCLSRC in the JSM library.

```
PGM PARM(&EVENT &INSTANCE &VERSION &JOB)
```

```
DCL VAR(&EVENT) TYPE(*CHAR) LEN(10)  
DCL VAR(&INSTANCE) TYPE(*CHAR) LEN(30)  
DCL VAR(&VERSION) TYPE(*CHAR) LEN(8)  
DCL VAR(&JOB) TYPE(*CHAR) LEN(10)
```

```
IF COND(&EVENT *EQ 'ENTRY') THEN(DO)  
  GOTO END  
ENDDO
```

```
IF COND(&EVENT *EQ 'REBOOT') THEN(DO)  
  GOTO END  
ENDDO
```

```
IF COND(&EVENT *EQ 'EXIT') THEN(DO)  
  GOTO END  
ENDDO
```

END: ENDPGM

2.1.4.2 Clear Java Service Manager (CLRJSM)

When tracing is enabled, tracing files are created in the trace subdirectory. This command is normally used to remove the old trace files and temporary files. For more details, refer to [Tracing](#).

The command will check that the instance path is valid by checking for the existence of following subdirectories.

- classes
- jar
- properties
- system
- temp
- trace

When you select the option to Clear the Java Service Manager, you will be prompted to enter the following:

INSTANCE

The instance defaults to a value of *DEFAULT. This is the recommended value. The default instance is defined in the JSMMGRDTA data area. This value can be changed using [2.1.4.6 Change Default Instance \(CHGJSMDFT\)](#).

TRACEDIR

The option defaults to a value of *YES. Valid values are *YES, *NO. A value of *YES will remove files and subdirectories in the trace directory. Refer to the KEEP keyword for information about selecting files to keep.

TEMPDIR

The option defaults to a value of *YES. Valid values are *YES, *NO. A value of *YES will remove all files and subdirectories in the temp directory. Do not clear the temp directory while JSM services are running, as any temporary files used by the services will be deleted. Use the value of *NO when clearing trace files from a running JSM instance.

KEEP

The option defaults to a value of 0. Valid values are in the range 0 to 99. A value of 0 means no files are kept and all files and subdirectories are deleted. A value of 1 means that only today's files are kept.

A value of 2 means that, today's and yesterday's files are kept.

2.1.4.3 Optimize Java Service Manager (OPTJSM)

It is no longer necessary to optimize JAR files. The new IBM Technology for Java JDKs do not use these optimized programs and the Classic JDKs have improved to such a point that these optimized programs do not offer much benefit.

This command will create optimized Java objects using [.jar](#) files located in the jar subdirectory and CLASS files located in the classes subdirectory. Optimization converts Java byte code into a hidden 64-bit RISC service program for the IBM i. This process may take several hours depending upon the speed of your IBM i server and the .jar file to be processed.

This command uses the i5/OS CRTJVAPGM command.

When you select the option to Optimize Java Service Manager, you will be prompted to enter the following:

INSTANCE

The instance defaults to a value of *DEFAULT. This is the recommended value. The default instance is defined in the JSMMGRDTA data area. This value can be changed using [2.1.4.6 Change Default Instance \(CHGJSMDFT\)](#).

OPTIMIZE

The option defaults to a value of 40. Valid values are 10, 20, 30, 40.

For more information about the OPTIMIZE keyword refer to the i5/OS CRTJVAPGM command.

JAR

The option defaults to *YES. Valid values are *NO, *YES. This indicates whether the files in the instance jar and endorsed subdirectories should be optimized.

CLASS

The option defaults to *YES. Valid values are *NO, *YES. This indicates whether the files in the instance classes subdirectory should be optimized.

REPLACE

The option defaults to *NO. Valid values include *NO, *YES. This indicates that an existing optimized program should be replaced.

For more information about the REPLACE keyword refer to the i5/OS

CRTJVAPGM command.

Important: Making Copies of Jar Files

If you make a copy of a .jar file for backup purposes, it is recommended that you do not leave the backup .jar file in the jar subdirectory. For example, before modifying jsn.jar, you copy the jsn.jar to oldjsn.jar in the same jar subdirectory. When the classpath is built using the list of files in the jar subdirectory, the oldjsn.jar will be included and might appear before jsn.jar in classpath. In this case, the old version could still be used, i.e.
classpath = oldjsn.jar;jsn.jar;jsmservice.jar;xerces.jar;xalan.jar...

2.1.4.4 Edit Manager Properties (EDTJSMMGR)

The Edit Manager Properties (EDTJSMMGR) command uses the EDTF command to edit the manager.properties file in the system subdirectory.

It is recommend you use Studio to maintain this file.

The manager.properties file configures the JSM instance. It controls the TCP ports used, tracing, secure socket layers and many other important properties of the JSM execution.

The manager.properties file is loaded when the JSM starts. If the manager properties change, you must stop and restart the JSM in order for the changes to take effect.

When you select the option to Edit JSM Manager Properties, you will be prompted to enter the following:

INSTANCE

The instance defaults to a value of *DEFAULT. This is the recommended value. The default instance is defined in the JSMMGRDTA data area. This value can be changed using [2.1.4.6 Change Default Instance \(CHGJSMDFT\)](#).

Following is an example of the manager.properties file:

```
Edit File: /jsm/instance/system/manager.properties
```

```
*****Beginning of data*****  
#  
# Java Service Manager configuration  
#  
tcp.port=4560  
tcp.backlog=20  
tcp.interface=*all  
#  
trace.manager=*yes  
# trace.transport.address=*all  
# trace.transport.error.address=*all  
# trace.service.address=*all  
# trace.service.error.address=*all  
#  
*****End of Data*****
```

Using a resource file allows the instance to be externally configured. The file can also be configured for different languages or regions. Refer to [Resource Properties-Internationalization](#).

2.1.4.5 Edit Service Properties (EDTJSMSRV)

The Edit Service Properties (EDTJSMSRV) command uses the EDTF command to edit the service.properties file in the system subdirectory.

It is recommend you use Studio to maintain this file.

The service.properties file configures the JSM services. It is a registry for the services used by the JSM. It is also used for controlling tracing and resources.

The service.properties file is loaded when the JSM starts. If the service properties change, you must refresh in order for the changes to take effect.

When you select the option to Edit JSM Service Properties, you will be prompted to enter the following:

INSTANCE

The instance defaults to a value of *DEFAULT. This is the recommended value. The default instance is defined in the JSMMGRDTA data area. This value can be changed using [2.1.4.6 Change Default Instance \(CHGJSMDFT\)](#).

Following is an example of the service.properties file:

Edit File: /jvm/instance/system/service.properties

```
*****Beginning of data*****
#
# Java Service Manager services
#
service.SMTPMailService=com.lansa.jsm.service.SMTPMailService
service.POP3MailService=com.lansa.jsm.service.POP3MailService
service.HTTPService=com.lansa.jsm.service.HTTPService
service.FTPService=com.lansa.jsm.service.FTPService
service.SQLService=com.lansa.jsm.service.SQLService
service.XMLQueueService=com.lansa.jsm.service.XMLQueueService
#
# trace.SMTPMailService=*all
# resource.SMTPMailService=SMTPMailService
#
*****End of Data*****
```

When the LANSA client requests that the JSM to load a service, only the name

of the service is specified.

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVICE_NAME')

When the JSM instance receives the SERVICE_LOAD command, the SMTPMailService servicename is prefixed with "service." The JSM searches the service.properties files. It uses the keyword SERVICE.SMTPMAILSERVICE to locate the class named com.lansa.jsm.service.SMTPMailService.

It is possible to associate a resource with this service by adding an entry with a "resource." prefix.

If a resource entry is present and the named resource file exists in the properties subdirectory, then this resource is passed to the service program via the JSMContainer getServiceResource() method.

If no resource entry exists then the service name is used as the resource file name.

If no resource file exists then an empty resource object is returned.

2.1.4.6 Change Default Instance (CHGJSMDFT)

When you select the option to Change Default Instance (CHGJSMDFT), you will be prompted to enter the following:

INSTANCE

Default instance path. This value is updated to the JSMMGRDTA data area in the JSM library. The LANSAs Integrator install will have correctly updated this data area.

Following is an example of this data area:

```
Value
Offset * ...+...1....+...2....+...3....+...4....+...5
0    '/jsm/instance'
```

WARNING: Be very careful when changing the default instance. Changing the instance can have a significant impact on your applications.

2.1.4.7 Work with Java Service Manager (WRKJSM)

Work with Java Service Manager (WRKJSM)

The Work with Java Service Manager (WRKJSM) command allows the WRKLNK command to display a tree view of the instance.

When you select the option to Work with Java Service Manager, you will be prompted to enter the following:

INSTANCE

The instance defaults to a value of *DEFAULT. This is the recommended value. The default instance is defined in the JSMMGRDTA data area. This value can be changed using [2.1.4.6 Change Default Instance \(CHGJSMDFT\)](#).

After you enter the instance, the [2.1.4.8 Work with Object Links](#) screen will be displayed.

2.1.4.8 Work with Object Links

This window displays a tree view of the instance. Following is an example of instance subdirectories:

Work with Object Links

Directory: /jsm/instance

Type options, press Enter.

2=Edit 3=Copy 4=Remove 5=Display 7=Rename
8=Display attributes 11=Change current directory ...

Opt	Object link	Type	Attribute	Text
	classes	DIR		
	dtd	DIR		
	jar	DIR		
	properties	DIR		
	system	DIR		
	temp	DIR		
	trace	DIR		
	xsl	DIR		

Parameters or command

===>

F3=Exit F4=Prompt F5=Refresh F9=Retrieve F12=Cancel
F17=Position to F22=Display entire field F23=More options

From this display, you can work with the files and directories for the Java Service Manager instance.

2.1.4.9 Change JSM Owner (CHGJSMOWN)

The Change JSM Owner (CHGJSMOWN) command will apply the CHGPGP and CHGOWN commands to all files and subdirectories including the JSM instance directory.

The command will check that the instance path is valid by checking for the existence of following subdirectories.

- classes
- jar
- properties
- system
- temp
- trace

When you enter the command:

CHGJSMOWN

and press F4 you will be prompted to enter the following:

NEWOWN

The name of the new owner of the files and directories below and including the JSM instance directory.

INSTANCE

The instance defaults to a value of *DEFAULT. This is the recommended value. The default instance is defined in the JSMMGRDTA data area. This value can be changed using [2.1.4.6 Change Default Instance \(CHGJSMDFT\)](#).

Example

```
CHGJSMOWN NEWOWN(QOTHPRDOWN) INSTANCE(*DEFAULT)
```

This program will call the CHGPGP and CHGOWN commands on all files and sub-directories in the instance directory.

```
CHGPGP OBJ(%s) NEWPGP(*NONE)
```

```
CHGOWN OBJ(%s) NEWOWN(%s)
```

Example from the DSPJOBLOG command

Object /jvm/instance changed.

Object /jvm/instance/classes changed.

Object /jvm/instance/dtd changed.

Object /jvm/instance/dtd/order.dtd changed.

...

2.1.4.10 Change JSM Authority (CHGJSMAUT)

The Change JSM Authority (CHGJSMAUT) command will apply the CHGAUT command to all files and subdirectories including the JSM instance directory.

The command will check that the instance path is valid by checking for the existence of following subdirectories.

- classes
- jar
- properties
- system
- temp
- trace

When you enter the command

CHGJSMAUT

and press F4 you will be prompted to enter the following:

USER

The name of the user whose authority is being set for all files and directories below and including the JSM instance directory.

DTAAUT

The level of data authority for all files and directories below and including the JSM instance directory.

OBJAUT

The level of object authority for all files and directories below and including the JSM instance directory.

INSTANCE

The instance defaults to a value of *DEFAULT. This is the recommended value. The default instance is defined in the JSMMGRDTA data area. This value can be changed using [2.1.4.6 Change Default Instance \(CHGJSMDFT\)](#).

Example

```
CHGJSMAUT USER(*PUBLIC) DTAAUT(*EXCLUDE) OBJAUT(*NONE)
```

This program will call the CHGAUT on all files and sub-directories in the instance directory.

CHGAUT USER(%s) DTAAUT(%s) OBJAUT(%s) OBJ(%s)

Example from the DSPJOBLOG command

Object /jsm/instance changed.

Object /jsm/instance/classes changed.

Object /jsm/instance/dtd changed.

Object /jsm/instance/dtd/order.dtd changed.

...

2.1.4.11 Change JSM Primary Group (CHGJSMPGP)

The Change JSM Primary Group (CHGJSMPGP) command will apply the CHGPGP command to all files and subdirectories including the JSM instance directory.

The command will check that the instance path is valid by checking for the existence of following subdirectories.

- classes
- jar
- properties
- system
- temp
- trace

When you enter the command

CHGJSMPGP

and press F4 you will be prompted to enter the following:

PGP

The name of the primary group whose authority is being set for all files and directories below and including the JSM instance directory.

DTAAUT

The level of data authority for all files and directories below and including the JSM instance directory.

OBJAUT

The level of object authority for all files and directories below and including the JSM instance directory.

INSTANCE

The instance defaults to a value of *DEFAULT. This is the recommended value. The default instance is defined in the JSMMGRDTA data area. This value can be changed using [2.1.4.6 Change Default Instance \(CHGJSMDFT\)](#).

Example

```
CHGJSMPGP PGP(*NONE) DTAAUT(*EXCLUDE) OBJAUT(*NONE) INS
```

This program will call the CHGPGP on all files and sub-directories in the instance directory.

CHGPGP NEWPGP(%s) DTAAUT(%s) OBJAUT(%s) OBJ(%s)

2.1.4.12 Change Attribute (CHGATR)

The Change Attribute command is an i5/OS command that allows the IFS file attributes to be set.

The most important attribute is READONLY.

The upgrade process will not be able to replace files that have the READONLY attribute set on.

Example

```
CHGATR OBJ('/jsm/instance') ATR(*READONLY) VALUE(*NO) SUBTREE
```

2.2 Java Service Manager Administration for Windows

The Java Service Manager Administration on Windows is a service application.

How To Start

On Windows, if the JSM service's Start-up Type is set to **Automatic**, Microsoft Service Control Manager will start the service automatically during system startup. If it is set to **Manual**, you need to use Microsoft Service Control Manager to start the JSM Administrator.

JSM Administrator starts the Java Virtual Machine and launches the Java Service Manager. If the administrator program has performed an upgrade, then the JSM will have stopped.

If the administrator program has stopped due to an upgrade, start the JSM Administrator again.

To stop the JSM server from ending when the user logs off, the JVM option -Xrs needs to be present.

Program Options

The JSM Administrator configuration options are specified in the jsmmgrdta.txt file.

Priority	The JVM process priority. Possible values are LOW, HIGH, NORMAL, BELOWNORMAL or ABOVENORMAL. Default is NORMAL.
Instance	The path of the instance. Default is ..\JSMInstance.
JVM	The path of the Java program to execute. To specify another JRE or JDK use the absolute path to the java program. For example: C:\Program Files\Java\jdk1.5.0_22\bin\java Default is java.
JVMOptions	The Java program options. There is no default value. The shipped value is -Xms128M -Xmx128M -Xrs -Djava.endorsed.dirs=.\endorsed

JSM is only compatible with a 32-bit JRE or JDK. Even if the JSM

Administrator is running on 64-bit Windows it still requires a 32-bit JRE or JDK to be installed.

Clear Trace

Clear Trace removes files and subdirectories in the trace and/or temp directories.

It can be run in interactive mode or as a batch job. The path of the instance is taken from the `jsmmgrdta.txt` file (see *Program Options for the Java Service Manager Administration for Windows*).

Clear Trace can be run from the Start | Programs | Menu or from the command line. You can use any of the following three options, all of which are optional.

/batch

Run Clear Trace as a batch job

/temp

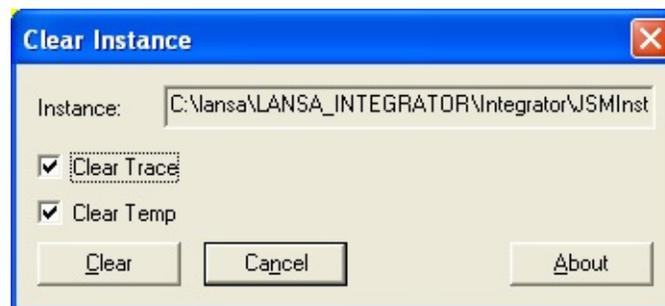
Remove files and subdirectories in the temp directory only.

/trace

Remove files and subdirectories in the trace directory only.

Interactive mode Example:

From the *Clear Instance* dialog, select the actions you wish Clear Trace to take:



This will result in:

clrjrm to remove trace and temp files (default behavior)

clrjrm /trace /temp to remove trace and temp files

clrjrm /temp to remove temp files and not trace files

clrjrm /trace to remove trace files and not temp files.

Batch mode Examples:

clrjrm/batch removes trace and temp files

clrjrm /batch /trace /temp removes trace and temp files

clrjrm /batch /temp removes temp files and not trace files

clrjrm /batch /trace removes trace files and not temp files.

2.3 Java Service Manager Administration for Linux

Starting the JSM

The Java Service Manager on Linux is started using the `strjasm` program found in `$LANSAXROOT/integrator/sh`. It should be run by the LANSAX owner, in its own shell. It can also be run as a background process.

After running `strjasm`, the following should be sent to standard output:

```
Starting JSM
Instance directory : /lansa/integrator/instance
Process ID       : 18621
```

Ending the JSM

You should use [Integrator Studio](#) to stop accepting clients to avoid disconnecting any clients in the middle of a transaction and to also shutdown the JSM.

Instance Location

By default, `strjasm` uses the `LANSAXROOT` environment variable to find the JSM instance to execute. The instance location, in this case, is `$LANSAXROOT/integrator/instance`. There are two ways to override this:

- Supply another instance location on the command line. For example, this code will use the instance found at `jsm/instance` in the current directory:

```
strjasm jsm/instance
```

- Set the `JSM_INSTANCE_DIR` environment variable to the new instance directory.

If `LANSAXROOT` and `JSM_INSTANCE_DIR` are both undefined and no instance directory is specified on the command line, `strjasm` will fail with an error.

Clear Trace Directory

If you run `strjasm` with the `-c` (lowercase c) option, it will clear the instance's trace directory before starting the JSM server. The trace directory will be empty

except for a directory for the current JSM server's process.

If the `JSM_CLEAR_TRACE` environment variable is set to a non-empty string, the trace directory will be cleared by default. In this case, it can be turned off with the `-C` (uppercase C) option.

2.4 Java Service Manager Console

Each Java Service Manager instance can have a HTTP web Console interface that allows a web browser or Studio application to perform administration tasks on the currently active service manager.

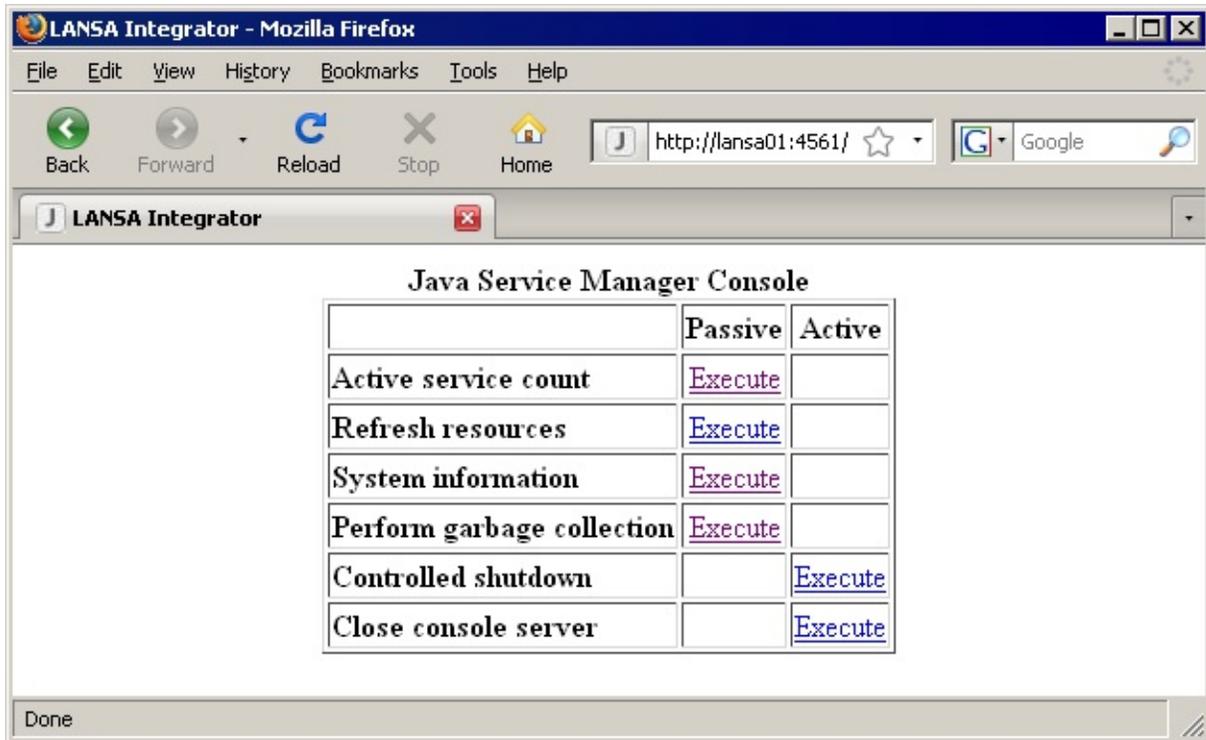
The following `manager.properties` entries control the Console address and port. If no Console interface is required comment out these entries in the `manager.properties` file.

```
console.tcp.port=4561
console.tcp.backlog=5
console.tcp.interface=*all
```

To control which clients can connect to the Console server use the following entries.

```
console.client.address=*all
studio.client.address=*all
```

To access the Console from a web browser, use the `http://host:port` URL.



The **Active service count** command returns the number of service threads currently running.

The **Refresh resources** command instructs the service manager to reload manager.properties, service.properties, structure.properties and list.properties files.

The **System information** command returns a table of information about the service manager environment.

The **Perform garbage collection** command instructs the service manager to perform a JVM garbage collection cycle.

The **Controlled shutdown** command instructs the service manager to stop accepting new JSM client connections. Currently running service threads will not be influenced. When no more service threads are running or when the shutdown wait time is reached the service manager ends.

The **Close console server** command ends the Console listener and no further Console and Studio connections are possible.

When the JSM instance starts and the Console interface has been enabled, a start entry is appended to the Console-log.txt file.

When a web browser or Studio application sends a request to the Console server thread, this request is written to the console-log.txt file in the system directory.

The console-log.txt entry is made up of a date time stamp, IP address of client

program and the request. This provides an audit log of all requests made using the Java Service Manager Console.

Example

```
[2005-02-03 01:57:56 +0000]
=====
[2005-02-03 01:58:50 +0000] [10.2.1.55] Request for download
[/jvm/instance/order.txt; ]
[2005-02-03 01:58:55 +0000] [10.2.1.55] Request for thread count
[2005-02-03 01:59:01 +0000] [10.2.1.55] Request for refresh
```

The console.timezone property can be used to adjust GMT to local time, if the Java 'user.timezone' has been set differently.

```
# A custom time zone like GMT+10:00 does not include daylight savings time
# console.timezone=GMT+10:00
# console.timezone=Australia/Sydney
console.timezone=AET
```

The console.authentication property controls basic authentication with the client web browser. If the console.authentication property has a value of *yes then client authentication is done.

```
# If console.authentication is commented out, then no authentication.
console.authentication=*yes
console.authentication.alick=6aae268520b50b6b4c28194631de5a24
```

If console authentication is turned on then client web browsers will prompt with an authentication dialog box.

A cached lookup of manager.properties for property console.authentication. {username} is done and the property value compared to the digest token created from the web browser authentication response.

If the user entry does not exist then the following will be logged.

```
[2005-04-01 03:04:37 +0000] No console authentication user : jack
```

If the user entry exists, but the digests do not match then the following will be logged.

```
[2005-04-01 03:02:31 +0000] Console authentication digest for alick is  
6aae268520b50b6b4c28194631de5a24
```

```
[2005-04-01 03:02:31 +0000] Client user has failed authentication check
```

The logged digest has been created by the client application, so update the `manager.properties` entry with this digest so they match on the next login attempt.

```
console.authentication.alick=6aae268520b50b6b4c28194631de5a24
```

If the user is authenticated then the log events include the user name

```
[2005-04-01 03:02:47 +0000] [10.2.1.55] [alick] GET /REQUEST-SYSTEM-  
REFRESH
```

The `console.client.address` can be used to control what web browser client addresses can connect, if more than one client address is needed then comma separate the multiple addresses.

```
console.client.address=*all  
# console.client.address=*none  
# console.client.address=10.2.1.7  
# console.client.address=10.2.1.7,10.2.1.8
```

The `studio.authentication` property controls authentication with the client Studio application. If the `studio.authentication` property has a value of `*yes` then client authentication is done.

```
# If studio.authentication is commented out, then no authentication.  
studio.authentication=*yes  
studio.authentication.alick=6aae268520b50b6b4c28194631de5a24
```

If studio authentication is turned on then client Studio applications will prompt

with an authentication dialog box.

A cached lookup of manager.properties for property studio.authentication.
{username} is done and the property value compared to the digest token send by
the client Studio application.

If the user entry does not exist then the following will be logged.

```
[2005-04-01 03:04:37 +0000] No studio authentication user : jack
```

If the user entry exists, but the digests do not match then the following will be
logged.

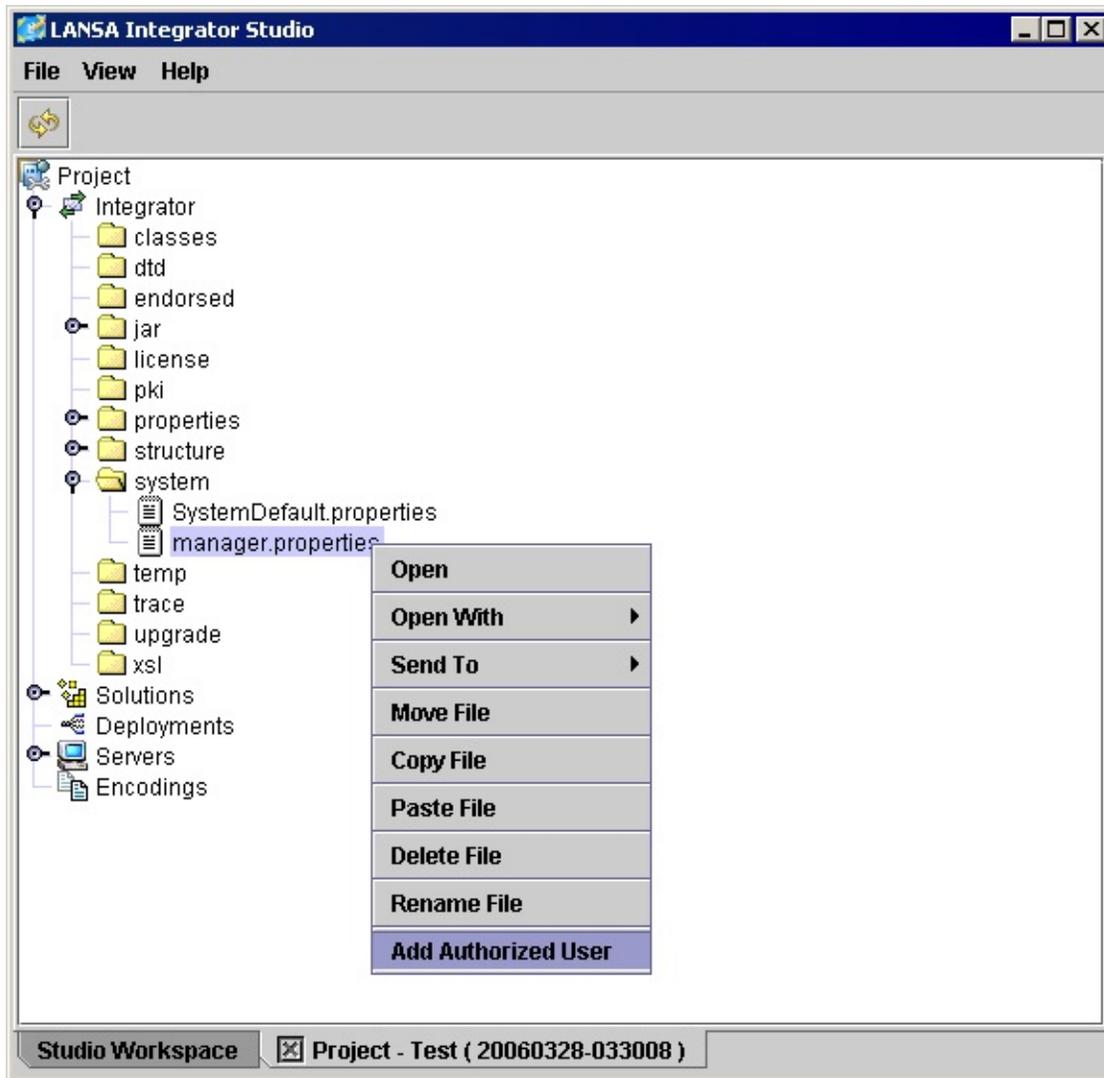
```
[2005-04-01 03:02:31 +0000] Studio authentication digest for alick is  
6aae268520b50b6b4c28194631de5a24
```

```
[2005-04-01 03:02:31 +0000] Client user has failed authentication check
```

Use Studio to create and add authorized users to the manager.properties section
file. Console and Studio users belong to the JSM realm.

```
studio.authentication.alick=6aae268520b50b6b4c28194631de5a24  
console.authentication.alick=6aae268520b50b6b4c28194631de5a24
```

Publish the modified manager.properties section file and use the instance refresh
menu item to reload the manager.properties file.



If the user is authenticated then the log events include the user name

```
[2005-04-01 03:03:23 +0000] [10.2.1.55] [alick] Request for download  
[/devjsm/instance/rehau1.xml;]
```

The studio.client.address can be used to control what studio client address can connect, if more than one client address is needed then comma separate the multiple addresses.

```
studio.client.address=*all  
# studio.client.address=*none  
# studio.client.address=10.2.1.7
```

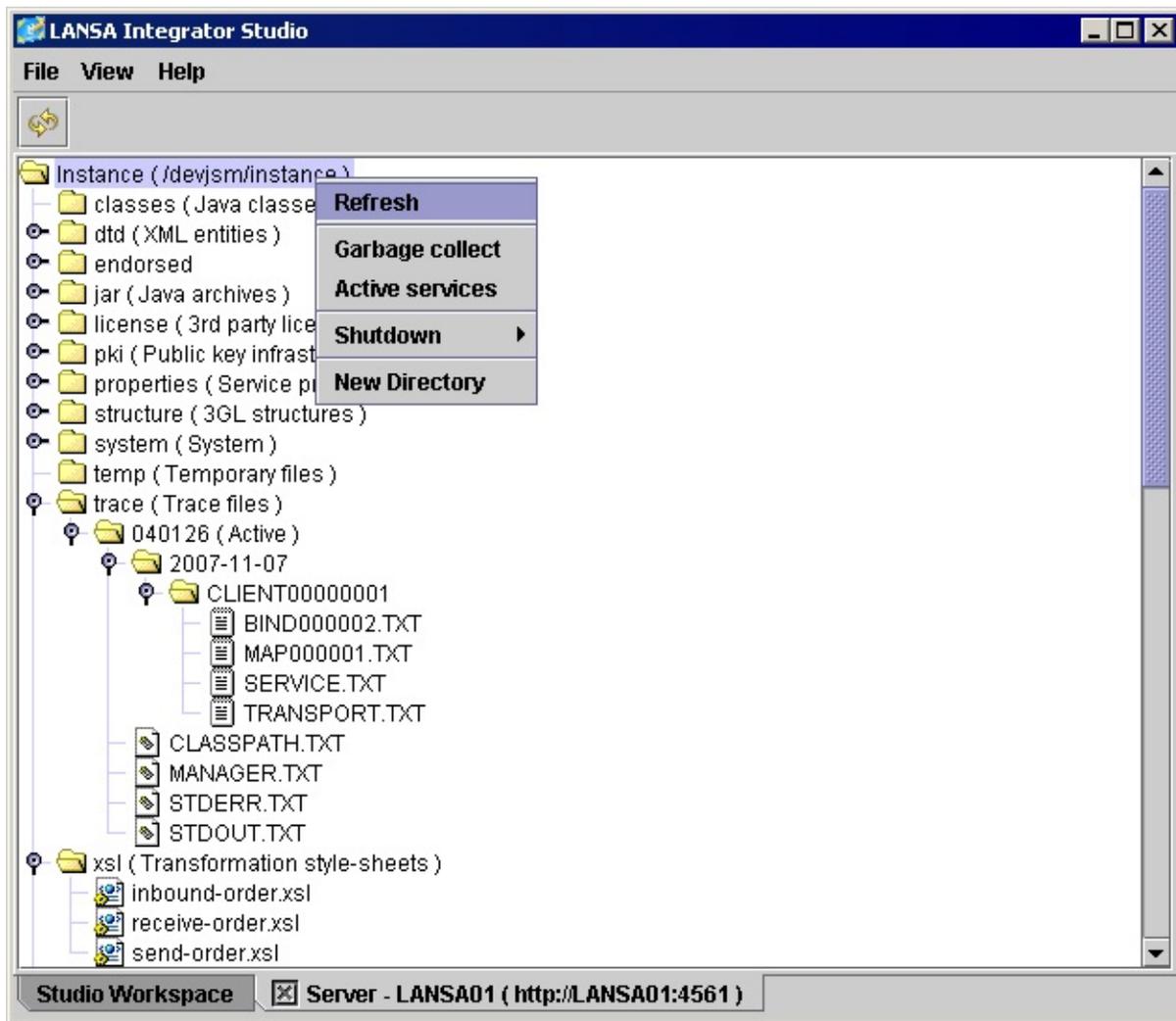
```
# studio.client.address=10.2.1.7,10.2.1.8
```

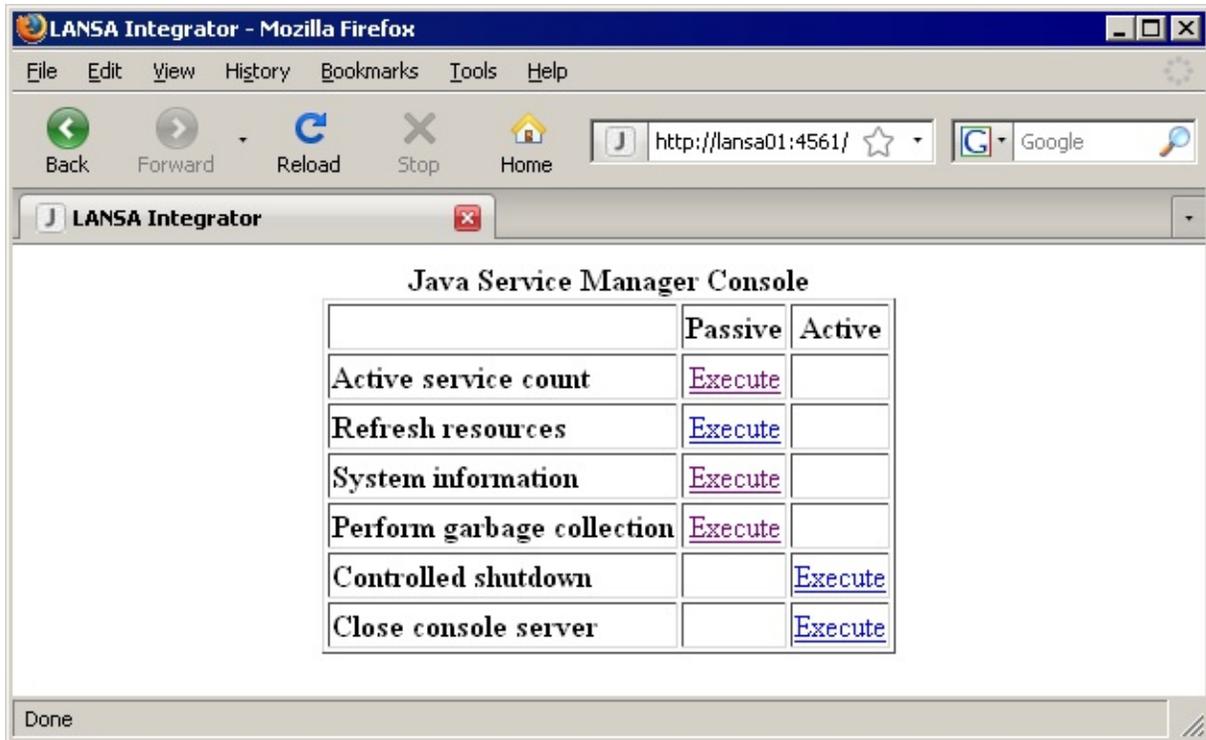
The `studio.include.trace.directory` can be used to control the depth of the trace directory being sent to the Studio client. By default all child directories in the trace directory are sent.

```
# studio.include.trace.directory=*all | *active | *none | *client
```

2.5 Java Service Manager Refresh

To refresh property files while the JSM is running use the Integrator Studio refresh menu item or Console Refresh resources option.





The following property files will be re-loaded:

- list.properties
- service.properties
- structure.properties
- manager.properties

Only the following manager properties will be available after a refresh:

- pool.hosts
- console.timezone
- console.authentication
- console.authentication.{user}
- studio.authentication
- studio.authentication.{user}
- studio.include.trace.directory
- shutdown.controlled.wait
- shutdown.controlled.repeat
- ftp.ssl.factory

- ftp.ssl.provider
- ftp.ssl.protocol
- ftp.ssl.keyStore
- ftp.ssl.keyStoreType
- ftp.ssl.keyStorePassword
- ftp.ssl.keyAlias
- ftp.ssl.keyAlgorithm
- ftp.ssl.trustStore
- ftp.ssl.trustStoreType
- ftp.ssl.trustStorePassword
- ftp.ssl.trustAlgorithm
- ftp.ssl.trustServer
- http.ssl.factory
- http.ssl.provider
- http.ssl.protocol
- http.ssl.keyStore
- http.ssl.keyStoreType
- http.ssl.keyStorePassword
- http.ssl.keyAlias
- http.ssl.keyAlgorithm
- http.ssl.trustStore
- http.ssl.trustStoreType
- http.ssl.trustStorePassword
- http.ssl.trustAlgorithm
- http.ssl.trustServer
- trace.threads

The following manager properties are used once at JSM startup and the refreshed value has no effect on the current environment:

- tcp.port
- tcp.backlog
- tcp.interface

- tcp.nodelay
- tcp.timeout
- tcp.buffer.send
- tcp.buffer.receive
- console.tcp.port
- console.tcp.backlog
- console.tcp.interface
- console.tcp.nodelay
- console.tcp.timeout
- console.tcp.buffer.send
- console.tcp.buffer.receive
- pool.tcp.port
- pool.tcp.backlog
- pool.tcp.interface
- pool.tcp.nodelay
- pool.tcp.timeout
- pool.tcp.buffer.send
- pool.tcp.buffer.receive
- manager.priority
- service.priority
- startup.class

The following manager properties are parsed and cached at JSM startup and the refreshed value has no effect on the current environment:

- tcp.client.address
- console.client.address
- studio.client.address
- trace.transport.address
- trace.transport.error.address
- trace.service.address
- trace.service.error.address
- trace.fields

- trace.passwords

The following manager properties are only set once at JSM startup and the refreshed value has no effect on the current environment:

- javax.net.debug
- javax.net.ssl.keyStore
- javax.net.ssl.keyStoreType
- javax.net.ssl.keyStorePassword
- javax.net.ssl.trustStore
- javax.net.ssl.trustStoreType
- javax.net.ssl.trustStorePassword
- ssl.KeyManagerFactory.algorithm
- ssl.TrustManagerFactory.algorithm
- ssl.SocketFactory.provider
- ssl.ServerSocketFactory.provider
- networkaddress.cache.ttl
- networkaddress.cache.negative.ttl
- os400.certificateLabel
- os400.certificateContainer
- log4j.configuration

2.6 Java Service Manager Pool Server

Each Java Service Manager instance can have a pool server interface. A pool server allows load balancing to be done. When a JSM client connects to a pool server the client is automatically redirected to one of the specified hosts in a round-robin process.

If a host:port entry specified in the pool.hosts list is not available, then the JSM client will go back to the pool server for another entry. The unavailable host machine must be running, so that the client TCP/IP socket connect attempt fails quickly and the client can return to the pool server. If the unavailable host machine is not running a long timeout delay will be experienced. This timeout delay can be up to 3 minutes on a IBM i machine.

In the following example the JSM client program connects to a pool server address instead of the JSM server address.

```
CHANGE  FIELD(#JSMSRV) TO("LOCALHOST:4562")
USE      BUILTIN(JSM_OPEN) WITH_ARGS(#JSMSRV) TO_GET(#JSMS
```

The following manager.properties entries control the pool server address and port. If no pool server interface is required comment out these entries in the manager.properties file.

```
# pool.tcp.port=4562
# pool.tcp.backlog=100
# pool.tcp.timeout=2000
# pool.tcp.interface=*all
# pool.hosts=HOST1:4560,HOST2:4560,HOST3:4560
```

The pool server uses the tcp.client.address property to control client connections.

2.7 Java Service Manager Additional Servers

Each Java Service Manager instance can support additional JSM servers.

This allows multiple JVM jobs to share the same JSM instance directory and files.

Use the same JDK version when sharing a JSM instance directory.

The STRJSM command submits the RUNJSM program which starts a JVM job to run the Java Service Manager.

The Java Service Manager uses the instance manager.properties file to determine its runtime configuration.

The Java Service Manager will attempt to bind to the base TCP/IP interface (tcp.interface and tcp.port) to start a JSM server.

If it cannot bind to the base TCP/IP interface it will attempt to bind to the next available additional TCP/IP interface (additional.tcp.interface.n and additional.tcp.port.n). Once it has bound to an additional TCP/IP interface, the optional console and pool server for that additional server are started.

It is possible to configure 1 to 10 additional servers.

The following trace examples illustrate how the first STRJSM command, starts a JSM server, a pool server and console server. The next STRJSM command using the same instance directory starts an additional JSM server.

Example: manager.properties

```
#
# Base instance
#
tcp.port=4560
tcp.backlog=20
# tcp.timeout=2000
tcp.interface=*all
#
console.tcp.port=4561
console.tcp.backlog=5
# console.tcp.timeout=2000
console.tcp.interface=*all
#
pool.tcp.port=4565
```

```
pool.tcp.backlog=20
# pool.tcp.timeout=2000
pool.tcp.interface=*all
# pool.hosts=LANSA01:7766,LANSA01:4560
# pool.hosts=LANSA01:7766,10.2.1.47:4560,LANSA01:4560
pool.hosts=LANSA06:4760
#
# Additional instance 1
#
additional.tcp.port.1=4360
additional.tcp.interface.1=*all
additional.tcp.backlog.1=25
# additional.console.port.1=4361
# additional.console.interface.1=*all
# additional.console.backlog.1=25
# additional.pool.port.1=4362
# additional.pool.interface.1=*all
# additional.pool.backlog.1=25
#
# additional.httpd.1=system/httpd-1.xml

#
# Additional instance 2
#
additional.tcp.port.2=4363
additional.tcp.interface.2=*all
additional.tcp.backlog.2=30
```

Example: First STRJSM MANAGER.TXT trace

```
manager: tcp.port      : 4560
manager: tcp.interface : *all
manager: tcp.backlog   : 20
manager: tcp.nodelay   : <null>
manager: tcp.buffer.send : <null>
manager: tcp.buffer.receive : <null>
manager: create manager server
manager: create socket address to listen on port 4560 across all interfaces
```

manager: bind to socket address
manager: start manager server
manager: server receive buffer size : 64000

manager: pool.tcp.port : 4565
manager: pool.tcp.interface : *all
manager: pool.tcp.backlog : 20
manager: pool.tcp.nodelay : <null>
manager: pool.tcp.buffer.send : <null>
manager: pool.tcp.buffer.receive : <null>
manager: create pool server
manager: create socket address to listen on port 4565 across all interfaces
manager: bind to socket address
manager: start pool server
manager: server receive buffer size : 64000
manager: pool host : LANSA06:4760

manager: console.tcp.port : 4561
manager: console.tcp.interface : *all
manager: console.tcp.backlog : 5
manager: console.tcp.nodelay : <null>
manager: console.tcp.buffer.send : <null>
manager: console.tcp.buffer.receive : <null>
manager: create console server
manager: create socket address to listen on port 4561 across all interfaces
manager: bind to socket address
manager: start console server
manager: server receive buffer size : 64000

Example: Second STRJSM MANAGER.TXT trace

manager: tcp.port : 4560
manager: tcp.interface : *all
manager: tcp.backlog : 20
manager: tcp.nodelay : <null>
manager: tcp.buffer.send : <null>
manager: tcp.buffer.receive : <null>

manager: create manager server
manager: create socket address to listen on port 4560 across all interfaces
manager: bind to socket address
manager: bind exception : Address already in use.

manager: additional manager server 1
manager: tcp.port : 4360
manager: tcp.interface : *all
manager: tcp.backlog : 25
manager: tcp.nodelay : <null>
manager: tcp.buffer.send : <null>
manager: tcp.buffer.receive : <null>
manager: create manager server
manager: create socket address to listen on port 4360 across all interfaces
manager: bind to socket address
manager: start manager server
manager: server receive buffer size : 64000
No additional pool server
No additional console server

2.8 Java Hotspot Technology

Refer to the following reference material:

[Java application launcher V1.5.0](#)

[Java application launcher V6](#)

[Java application launcher V7](#)

Java command arguments

Argument	Description
-Xms<size>	Specify the initial heap size, in bytes, of the memory allocation pool. This value must be a multiple of 1024 greater than 1MB. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, g or G to indicate gigabytes.
-Xmx<size>	Specify the maximum heap size, in bytes, of the memory allocation pool. This value must be a multiple of 1024 greater than 2MB. Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, g or G to indicate gigabytes.
-Xss<size>	Specify the thread stack size.
-Xrs	Reduces usage of operating-system signals.

Example

```
java -Xms1G -Xmx1G -Xrs ...
```

2.9 Java Endorsed Standards Override Mechanism

An endorsed standard is a Java API defined through a standards process other than the Java Community Process (JCP).

Because endorsed standards are defined outside the JCP, it is anticipated that such standards may be revised between releases of the Java 2 Platform.

In order to take advantage of new revisions to endorsed standards, developers and software vendors may use the Endorsed Standards Override Mechanism to provide newer versions of an endorsed standard than those included in the Java 2 Platform as released by Sun Microsystems.

Classes implementing newer versions of endorsed standards should be placed in JAR files. The system property 'java.endorsed.dirs' specifies one or more directories that the Java runtime environment will search for such JAR files.

Refer to: <http://download.oracle.com/javase/1.5.0/docs/guide/standards/>

Issues running Xalan-Java on JDK 1.4

Some versions of JDK 1.4 are packaged with an old version (2.2D11) of Xalan-Java.

JDK 1.4 will attempt to use this version instead of any on the classpath.

Unfortunately, this causes problems when attempting to use a newer version of Xalan-Java with the JDK 1.4.

Java command arguments

Argument	Description
-Djava.endorsed.dirs	Directory containing the latest Xalan jar files.

2.10 JSM Startup Class

When JSM starts an internal startup class runs. This class loads all JSM service classes and performs an XML transformation to load as many Apache Xalan classes as possible.

A user-defined startup class can be specified by the 'startup.class' property in the manager.properties file.

```
startup.class=com.acme.MyStartup
```

The user-defined startup class needs to implement the Runnable interface. JSM casts this loaded class to a Runnable object and then executes the run method. This happens in the main thread and no additional threads are created to do this task.

The internal startup class and the user-defined startup class run before the JSM starts to accept client and console connections.

Example 1

```
package com.acme ;

public final class MyStartup implements Runnable
{
    public void run ()
    {
        try
        {

            /* YOUR CODE GOES HERE */

        }
        catch ( Throwable t )
        {
            t.printStackTrace () ;
        }
    }
}
```

Example 2

```
package com.acme ;

public final class MyStartup implements Runnable
{
    private int m_sleepTime = 0 ;

    public MyStartup ()
    {
        /*
         * JSMMManager uses the zero argument constructor
         */

        int seconds = 60 * 20 ; // Every 20 minutes

        Thread thread = new Thread ( new MyStartup ( seconds ) ) ;

        thread.start () ;
    }

    public MyStartup ( int seconds )
    {
        /*
         * Specify sleep time
         */

        if ( seconds <= 0 )
        {
            seconds = 0 ;
        }

        m_sleepTime = seconds * 1000 ;
    }

    public void run ()
    {
        if ( m_sleepTime == 0 )
```

```
{
    /*
       JSManager call
    */

    System.out.println ( "JSM warmup call" );

    try
    {
        warmup () ;
    }
    catch ( Throwable t )
    {
        t.printStackTrace () ;
    }

    return ;
}

/*
   MyStartup call with sleep time
*/

while ( true )
{
    try
    {
        Thread.sleep ( m_sleepTime ) ;

        System.out.println ( "MyStartup repeat warmup call" ) ;

        warmup () ;
    }
    catch ( Throwable t )
    {
        t.printStackTrace () ;
    }
}
```

```
}  
  
private final void warmup () throws Exception  
{  
    System.out.println ( "MyStartup warmup" );  
  
    /* YOUR CODE GOES HERE */  
  
}  
}
```

2.11 Activation Framework

When JSM starts the Activation Framework default command map and default file type map are set using the mailcap.txt and filetype.txt files located in the system subdirectory.

SMTP/POP3, AS2/AS3 and HTTP services use the command map to encode and decode MIME content.

The HTTP content handlers, InboundFileHandler, OutboundFileHandler and OutboundMultiPartHandler use the file type map to associate filename extension to MIME type.

2.12 IBM Java System Properties

The 'user.home' property is set to the JSM instance system directory. The SystemDefault.properties file located in the JSM instance system directory can be used to set Java system properties.

i5/OS and the JVM determine the values for Java system properties by using the following order of precedence:

- Command line
- QIBM_JAVA_PROPERTIES_FILE environment variable
- user.home SystemDefault.properties file
- /QIBM/UserData/Java400/SystemDefault.properties
- Default system property values

The IBM Technology for Java JVM's read the SystemDefault.properties file like the Classic JVM's.

If the first line of the SystemDefault.properties file starts with #AllowOptions, then JVM options and System properties can be used.

```
#AllowOptions
#
#!<studio-project id="20000000-000000" name="lansa">
# ...
#!</studio-project>
#
#!<studio-project id="20080101-000000" name="CustomChanges">
#
# JVM options
#
-verbose:sizes
-Xgcpolicy:gencon
# -Xcompactexplicitgc
# -Xtgc:excessiveGC,compaction
# -Xverbosegclog:GC-%Y-%m-%d-%H%M%S.XML
# -Xcompressedrefs
#
# System properties
#
```

```
java.awt.headless=true
user.timezone=GMT+10:00
os400.dir.create.auth=none
os400.file.create.auth=none
#
#!</studio-project>
#
```

2.13 IBM Technology for Java Shared Classes

Class sharing in the IBM Technology for Java SDK's offers a transparent and dynamic means of sharing all loaded classes, both application classes and system classes, and placing no restrictions on JVMs that are sharing the class data (unless runtime bytecode modification is being used).

Sharing all immutable class data for an application between multiple JVMs has obvious benefits:

- The virtual memory footprint reduction when using more than one JVM instance can be significant.
- Loading classes from a populated cache is faster than loading classes from disk, because the classes are already in memory and are already partially verified.

Therefore, class sharing also benefits applications that regularly start new JVM instances doing similar tasks.

The cost to populate an empty cache with a single JVM is minimal and, when more than one JVM is populating the cache concurrently, this activity is typically faster than both JVMs loading the classes from disk.

Key points to note about the IBM class sharing feature are as follows.

- Classes are stored in a named "class cache", which is either a memory-mapped file or an area of shared memory, allocated by the first JVM that needs to use it.
- Any JVM can read from or update the cache, although a JVM can connect to only one cache at a time.
- The cache persists beyond the lifetime of any JVM connected to it, until it is explicitly destroyed or until the operating system is shut down.
- When a JVM loads a class, it looks first for the class in the cache to which it is connected and, if it finds the class it needs, it loads the class from the cache.
- Otherwise, it loads the class from disk and adds it to the cache where possible.
- When a cache becomes full, classes in the cache can still be shared, but no new data can be added.
- Because the class cache persists beyond the lifetime of any JVM connected to it, if changes are made to classes on the file system, some classes in the

cache might become out of date (or "stale"). This situation is managed transparently; the updated version of the class is detected by the next JVM that loads it and the class cache is updated where possible.

- Sharing of bytecode that is modified at runtime is supported, but must be used with care.
- Access to the class data cache is protected by Java permissions if a security manager is installed.
- Classes generated using reflection cannot be shared.
- Only class data that does not change can be shared. Resources, objects, JIT compiled code, and similar items cannot be stored in the cache.

Creating shared classes

You switch on shared classes with the SystemDefault.properties `-Xshareclasses` and `-Xscmx` command-line options.

`-Xscmx<size>` Specifies cache size. This option applies only if a cache is being created and no cache of the same name exists. Default cache size is platform-dependent. You can find out the size value being used by adding `-verbose:sizes` as a command-line argument. Minimum cache size is 4 KB. Maximum cache size is platform-dependent.

The size of cache you can specify is limited by the amount of physical memory and paging space available to the system. Because the virtual address space of a process is shared between the shared classes cache and the Java heap, increasing the maximum size of the Java heap will reduce the size of the shared classes cache you can create.

`-Xshareclasses:<suboptions>` Enables class sharing. Can take a number of suboptions, some of which are cache utilities. Cache utilities perform the required operation on the specified cache, without starting the VM. You can combine multiple suboptions, separated by commas, but the cache utilities are mutually exclusive.

`#AllowOptions`

`-Xscmx16M`

`-Xshareclasses:name=myJSM`

Listing shared classes

When listing caches, you must use the same JVM as the shared class cache was

created for, except versions JDK 6 which can list all version caches.

With JDK5, if the JVM listing the shared classes does not find a compatible shared class, it exits with the JVMSHRC005I message regardless of what shared classes exist.

With JDK6, the VM will usually recognize shared class caches from other J9 VM's but will list them as an incompatible shared class.

Hint: Instead of setting the JAVA_HOME environment variable, use the absolute path to the JDK java shell script file.

QSH

```
/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit/bin/java -  
Xshareclasses:listAllCaches
```

Listing all caches in cacheDir /tmp/javasharedresources/

Cache name	level	persistent	last detach time
------------	-------	------------	------------------

Compatible shared caches

myJSM	Java6 64-bit	yes	In use
-------	--------------	-----	--------

Incompatible shared caches

myJSM	Java5 64-bit	no	Wed Jun 25 14:49:44 2008
-------	--------------	----	--------------------------

myJSM	Java5 32-bit	no	Wed Jun 25 14:46:06 2008
-------	--------------	----	--------------------------

myJSM	Java6 32-bit	yes	Wed Jun 18 12:18:59 2008
-------	--------------	-----	--------------------------

Note: There are four myJSM caches, one for each of the STRJSM JVM versions that had been started.

Destroying shared classes

When destroying caches, you must use the same JVM as the shared class cache was created for.

QSH

```
/QOpenSys/QIBM/ProdData/JavaVM/jdk50/64bit/bin/java -  
Xshareclasses:destroy,name=myJSM
```

```
JVMSHRC010I Shared Cache "myJSM" is destroyed
```

Unable to create Java Virtual Machine.

2.14 LOG4J Logging Services

Several open source products used by LANSA Integrator use the Apache LOG4J logging services.

If you need to log a product that supports LOG4J, then add two lines to the log4j.properties file, one to turn on a logger and the other to stop log messages going up the hierarchy to the root logger.

```
log4j.logger.xyz=DEBUG, FILE
log4j.additivity.xyz=false
```

manager.properties

```
log4j.configuration=system/log4j.properties
```

log4j.properties

```
#!<studio-project id="20000000-000000" name="lansa">
#
# This file controls the logging strategy for the JSM server
#
# Levels - DEBUG, INFO, WARN, ERROR, FATAL
#
log4j.debug=true
log4j.rootLogger=INFO, STDOUT
#
# log4j.logger.org.apache.axis=DEBUG, FILE
# log4j.additivity.org.apache.axis=false
#
# log4j.logger.org.apache.commons.httpclient=DEBUG, FILE
# log4j.additivity.org.apache.commons.httpclient=false
#
# log4j.logger.org.apache.axis.transport.http.HTTPSender=DEBUG, FILE
# log4j.additivity.org.apache.axis.transport.http.HTTPSender=false
#
log4j.appender.STDOUT=org.apache.log4j.ConsoleAppender
```

```
log4j.appender.STDOUT.threshold=INFO
log4j.appender.STDOUT.layout=org.apache.log4j.PatternLayout
log4j.appender.STDOUT.layout.ConversionPattern=
[%d{ISO8601}] [%-5p] [%c] - %m%n
#
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.file=LOG4J.TXT
log4j.appender.FILE.encoding=UTF-8
log4j.appender.FILE.append=false
log4j.appender.FILE.threshold=DEBUG
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=
[%d{ISO8601}] [%-5p] [%c] - %m%n
#
#!</studio-project>
```

2.15 Axis Properties

When the JSM Manager starts the name value entries in the system/AxisDefault.properties file are read and added to the org.apache.axis.AxisProperties class using the static method setProperty. The following is an example AxisDefault.properties file.

```
#!<studio-project id="20000000-000000" name="lansa">
#
# Axis default properties
#
axis.ClientConfigFile=system/axis-client-config.xml
axis.ServerConfigFile=system/axis-server-config.xml
#
# axis.http.client.maximum.total.connections
# axis.http.client.maximum.connections.per.host
# axis.http.client.connection.pool.timeout
# axis.http.client.connection.default.so.timeout
# axis.http.client.connection.default.connection.timeout
# axis.socketFactory
# axis.socketSecureFactory
# axis.ServerFactory
# http.proxyHost
# http.proxyPort
# http.proxyUser
# http.proxyPassword
# http.nonProxyHosts
# https.proxyHost
# https.proxyPort
# https.proxyUser
# https.proxyPassword
# https.nonProxyHosts
#
#!</studio-project>
```

Axis Global Configuration

Axis Reference Guide

Axis Client and Server Configuration

By default, Axis uses the 'org/apache/axis/client/client-config.wsdd' and 'org/apache/axis/server/server-config.wsdd' files from the jsmaxis.jar file for client and server configuration.

The AxisDefault.properties entries axis.ClientConfigFile and axis.ServerConfigFile direct Axis to use different configuration files.

```
axis.ClientConfigFile=system/axis-client-config.xml  
axis.ServerConfigFile=system/axis-server-config.xml
```

Axis Client Configuration

```
<?xml version="1.0" encoding="utf-8"?>  
  
<deployment name="defaultClientConfiguration" xmlns="http://xml.apache.org/axis/wsdd/">  
  
  <globalConfiguration>  
    <parameter name="disablePrettyXML" value="true"/>  
    <parameter name="addressing.sendReplyTo" value="true"/>  
    <parameter name="enableNamespacePrefixOptimization" value="false"/>  
  </globalConfiguration>  
  
  <transport name="http"  
    pivot="java:org.apache.axis.transport.http.HTTPSender"/>  
  
</deployment>
```

Axis Server Configuration

```
<?xml version="1.0" encoding="utf-8"?>  
  
<deployment name="defaultServerConfiguration" xmlns="http://xml.apache.org/axis/wsdd/">  
  
  <globalConfiguration>  
    <parameter name="sendMultiRefs" value="false"/>  
    <parameter name="dotNetSoapEncFix" value="true"/>  
  </globalConfiguration>  
  
</deployment>
```

```
<parameter name="disablePrettyXML" value="true"/>
<parameter name="enableNamespacePrefixOptimization" value="false"/>
</globalConfiguration>

</deployment>
```

Turning off MultiRef encoding in SOAP server responses

The axis server configuration can disable sending multiRefs in RPC/encoded responses by using the global sendMultiRefs parameter.

```
<parameter name="sendMultiRefs" value="false"/>
```

Using Commons HTTP client with Axis

By default Apache Axis 1.4 uses 'org.apache.axis.transport.http.HTTPSender' for http sending.

It is possible to change this to use 'org.apache.commons.httpclient'.

Change the transport pivot attribute in the configuration to point to the CommonsHTTPSender class.

```
<transport name="http" pivot="java:org.apache.axis.transport.http.CommonsH
```

You also need to add the common.codec.1.3.jar and common-httpclient-3.0-rc4.jar to the jar directory.

<http://jakarta.apache.org/commons/>

<http://jakarta.apache.org/commons/httpclient/>

<http://jakarta.apache.org/commons/httpclient/features>

<http://jakarta.apache.org/commons/httpclient/logging>

http://jakarta.apache.org/site/downloads/downloads_commons

Problems with using Commons HTTP client with JSMDirect server

By default commons http client uses HTTP protocol 1.1 and by default it uses chunked transfer encoding.

When a HTTP client program sends Transfer-Encoding chunked, it cannot send Content-Length.

JSMDirect is expecting Content-Length, so it can read STDIN to send to the JSM.

So the JSM SOAP server service receives no SOAP message content.

The SOAP Agent Wizard can force the http client program to use the HTTP 1.0 protocol by including the following code into the generated service code.

```
stub._setProperty ( org.apache.axis.MessageContext.HTTP_TRANSPORT_VE
                    org.apache.axis.transport.http.HTTPConstants.HEADER_PROTO
```

Including SOAP headers in SOAP request

It is possible to include a SOAP header in the SOAP request by including code in the generated SOAP agent service code. When the SOAP Agent Wizard generates the service code the contents of file AGENT_INCLUDE.TXT are included into the generated code.

It is also possible to add a SOAP header to a message by using a message handler. Refer to [2.17 SOAP Agent Message Handler](#).

```
/*
  Add SOAP header
*/
org.apache.axis.message.SOAPHeaderElement elementHead = new org.apache
;

javax.xml.soap.SOAPElement elementUserToken = elementHead.addChildEle

javax.xml.soap.SOAPElement elementUserName = elementUserToken.addChi
elementUserName.addTextNode ( "username" );

javax.xml.soap.SOAPElement elementPassword = elementUserToken.addChil
elementPassword.addTextNode ( "password" );

stub.setHeader ( elementHead );
```


2.16 Axis Message Handler

Refer to [2.17 SOAP Agent Message Handler](#).

Apache Axis allows custom message handlers to be created and deployed.

These handlers allow additional processing on the SOAP XML messages before transmission.

To determine the configuration name of your service, turn on LOG4J logging and run the SOAP Agent service once.

```
log4j.logger.org.apache.axis=DEBUG, FILE  
log4j.additivity.org.apache.axis=false
```

Search the LOG4J.TXT file for the setTargetService string. The value within the brackets is the configuration service name.

```
[DEBUG] [message.SOAPBody] addBodyElement - Adding body element to 1  
[DEBUG] [client.Call] invoke - Enter: Call::invoke()  
[DEBUG] [axis.MessageContext] setTargetService - MessageContext: setTarg
```

Add a deployment service entry to the configuration file.

The handler type value is the custom Java handler class.

It is also possible to pass configuration parameters to the handler instance.

```
<service name="MyServicePort">  
  <requestFlow>  
    <handler type="java:com.acme.axis.handler.MyHandler">  
      <parameter name="acme.keyword" value="value"/>  
    </handler>  
  </requestFlow>  
</service>
```

The custom handler class needs to extend the 'org.apache.axis.handlers.BasicHandler' class.

```
package com.acme.axis.handler ;

import org.apache.axis.AxisFault ;
import org.apache.axis.MessageContext ;

import org.apache.axis.message.PrefixedQName ;

import org.apache.axis.handlers.BasicHandler ;

import org.w3c.dom.Node ;
import org.w3c.dom.NodeList ;

import javax.xml.soap.Name ;
import javax.xml.soap.SOAPPart ;
import javax.xml.soap.SOAPBody ;
import javax.xml.soap.SOAPElement ;
import javax.xml.soap.SOAPMessage ;
import javax.xml.soap.SOAPEnvelope ;
import javax.xml.soap.SOAPException ;

import javax.xml.rpc.handler.soap.SOAPMessageContext ;

public class MyHandler extends BasicHandler
{
    public void invoke ( MessageContext messageContext ) throws AxisFault
    {
        System.out.println ( "MyHandler: invoke" ) ;

        String value = (String)getOption ( "acme.keyword" ) ;

        modifyMessage ( messageContext ) ;
    }
}
```

2.17 SOAP Agent Message Handler

The SOAP Agent message handler framework was developed to over-come limitations in the Axis Message handler framework and is the preferred way to add request and response handlers.

Message handlers developed for Apache Axis Message handler framework can be used in SOAP Agent message handler framework.

To add a message handler to a particular SOAP Agent service add a 'service.handlers.servicename' entry to the SOAPAgentService.properties file.

When the SOAP Agent command OPEN SERVICE (servicename) is executed the service class and message handler file are assigned to the executing program.

```
service.test=com.acme.service.soap.TestService  
service.handlers.test=handlers/soapagent-handlers.xml
```

More than one agent service configuration can be included in a single file.

It is optional to add a request or response handler.

A simple message handler chain can be created, by including more than one handler element.

Request and response message handlers can be assigned to all service operations or one particular operation.

Each handler element can be configured with zero or more parameter elements.

These parameter elements are instantiated and passed to the instantiated handler class.

If the parameter value attribute contains a value within open and close curly brackets, then the value is assumed to be a LANSAs field name and the value of the LANSAs field is passed to the handler. Use the

SERVICE_EXCHANGE(*FIELD) keyword on the CALL command to make LANSAs fields available to the handler class.

Message handler configuration

```
<?xml version="1.0" encoding="utf-8"?>
```

```

<services>

<!-- Assign SOAPHeaderHandler to all operations in the Test service -->
<service name="test">

  <request>
    <handler class="com.acme.axis.handler.SOAPHeaderHandler"/>
  </request>

  <response>
    <handler class="com.acme.axis.handler.SOAPHeaderHandler"/>
  </response>

</service>

<!--
- Assign SecurityHandler and SOAPHeaderHandler to the Test service login op
-->
<service name="test" operation="login">

  <request>
    <handler class="com.acme.axis.handler.SecurityHandler">
      <parameter name="user" value="{USER}"/>
      <parameter name="acme.keyword" value="ABC"/>
    </handler>
    <handler class="com.acme.axis.handler.SOAPHeaderHandler"/>
  </request>

</service>

</services>

```

Example Message Handler Class

```
package com.acme.axis.handler ;
```

```
import java.io.* ;

import java.util.Vector ;

import org.apache.axis.Message ;
import org.apache.axis.SOAPPart ;
import org.apache.axis.AxisFault ;
import org.apache.axis.MessageContext ;

import org.apache.axis.utils.XMLUtils ;

import org.apache.axis.message.SOAPBody ;
import org.apache.axis.message.SOAPHeader ;
import org.apache.axis.message.SOAPEnvelope ;
import org.apache.axis.message.SOAPBodyElement ;
import org.apache.axis.message.SOAPHeaderElement ;
import org.apache.axis.message.MessageElement ;

import org.apache.axis.message.RPCParam ;
import org.apache.axis.message.RPCElement ;
import org.apache.axis.message.PrefixedQName ;

import org.apache.axis.description.ParameterDesc ;

import org.apache.axis.encoding.SerializationContext ;

import org.w3c.dom.Node ;
import org.w3c.dom.Element ;
import org.w3c.dom.Document ;
import org.w3c.dom.NodeList ;

import org.w3c.dom.ls.LSOutput ;
import org.w3c.dom.ls.LSSerializer ;
import org.w3c.dom.ls.DOMImplementationLS ;

import org.w3c.dom.bootstrap.DOMImplementationRegistry ;

import com.lansa.jsm.JSMTrace ;
import com.lansa.jsm.JSMCommand ;
```

```

import com.lansa.jsm.JSMResource ;

import com.lansa.jsm.service.ServiceHelper ;

public class MyHandler extends org.apache.axis.handlers.BasicHandler
{
    private final static String EMPTY_STRING = "" ;

    private final static String ENCODING_UTF8 = "UTF-8" ;

    private final static String[] FORM_NAMES = { "", "FORM_STRING", "FC

    private JSMTrace m_trace = null ;

    private JSMResource m_serviceResource = null ;

    public void init ()
    {
        /*
            Optional - this over-rides init stub in BasicHandler

            Firstly, init is called on all handlers

            Secondly, invoke is called on all handlers

            Finally, cleanup is called on all handlers

            You could pass information between handlers by using:

            MessageContext - setProperty ( String name, Object value )

            MessageContext - Object value getProperty ( String name )
        */
    }

    public void invoke ( org.apache.axis.MessageContext messageContext ) thro
    {
        try
        {

```

```

    m_trace = (JSMTrace)getOption ( "jsm.handler.property.trace" );

    m_serviceResource = (JSMResource)getOption ( "jsm.handler.property

traceOptions () ;

    modifyMessage ( messageContext ) ;
}
catch ( Exception e )
{
    throw new AxisFault ( "MyHandler: exception : " + e.toString () ) ;
}
}

public void cleanup ()
{
    /*
    Optional - this over-rides cleanup stub in BasicHandler
    */

    if ( m_trace != null )
    {
        m_trace.println ( "MyHandler: cleanup" ) ;
    }
}

private final void traceOptions ()
{
    /*
    The following properties are from the handler parameters
    */

    String value = (String)getOption ( "acme.keyword" ) ;

    /*
    The following properties are supplied by the JSM service
    */

    String type = (String)getOption ( "jsm.handler.property.type" ) ;

```

```

String service = (String)getOption ( "jsm.handler.property.service" );

String operation = (String)getOption ( "jsm.handler.property.operation" );

/*
   CALL command
*/

JSMCommand command = (JSMCommand)getOption ( "jsm.handler.pro

if ( m_trace != null )
{
    m_trace.println ( "MyHandler: invoke" );

    m_trace.println ( "MyHandler: acme.keyword : " + value );

    m_trace.println ( "MyHandler: jsm.handler.property.type : " + type );

    m_trace.println ( "MyHandler: jsm.handler.property.service : " + servic
    m_trace.println ( "MyHandler: jsm.handler.property.operation : " + ope
    m_trace.println ( "MyHandler: jsm.handler.property.command : " + con
}
}

private final void modifyMessage ( org.apache.axis.MessageContext messag
{
    Message message = messageContext.getCurrentMessage () ;

    if ( message == null )
    {
        throw new IllegalArgumentException ( "no message available" );
    }

    /*
       Trace message - output is dependent on internal form
    */

```

```

//    traceMessage ( message ) ;

    /*
        SOAP Part
    */

    SOAPPart soapPart = (SOAPPart)message.getSOAPPart () ;

    int form = soapPart.getCurrentForm () ;

    if ( m_trace != null )
    {
        m_trace.println ( "MyHandler: current message form : ", FORM_NAM
    }

    /*
        SOAP Envelope - the internal form is converted to SOAPEnvelope
    */

    SOAPEnvelope envelope = soapPart.getAsSOAPEnvelope () ;

//    traceEnvelope ( envelope ) ;

    /*
        Handle request or response
    */

    if ( isRequest () )
    {
        modifyHeader ( envelope ) ;

        traceBody ( envelope ) ;

//        setRequestMessage1 ( soapPart ) ;

//        setRequestMessage2 ( messageContext ) ;

        setRequestMessage3 ( messageContext, envelope ) ;

```

```

        return ;
    }

    if ( isResponse () )
    {
        setResponseMessage1 ( messageContext ) ;

        return ;
    }
}

private final void modifyHeader ( SOAPEnvelope envelope ) throws Except
{
    Vector vector = envelope.getHeaders () ;

    int count = vector.size () ;

    if ( m_trace != null )
    {
        m_trace.println ( "MyHandler: header count : " + count ) ;
    }

    for ( int i=0; i < count; i++ )
    {
        SOAPHeaderElement element = (SOAPHeaderElement)vector.elementAt

        if ( m_trace != null )
        {
            m_trace.println ( "MyHandler: soap header : " + element.getElement

        }
    }

    if ( m_trace != null )
    {
        m_trace.println ( "MyHandler: add header" ) ;
    }

    envelope.addHeader ( createHeader () ) ;
}

```

```
}
```

```
private final SOAPHeaderElement createHeader () throws Exception
```

```
{
```

```
    SOAPHeaderElement elementHead = new org.apache.axis.message.SOAPHeaderElement (
```

```
        MessageElement elementSession = (MessageElement)elementHead.addContent (
```

```
            elementSession.addTextNode ( "text" );
```

```
            elementHead.setActor ( null );
```

```
        return elementHead ;
```

```
}
```

```
private final boolean isRequest ()
```

```
{
```

```
    String type = (String)getOption ( "jsm.handler.property.type" );
```

```
    if ( type.equals ( "request" ) )
```

```
    {
```

```
        return true ;
```

```
    }
```

```
    return false ;
```

```
}
```

```
private final boolean isResponse ()
```

```
{
```

```
    String type = (String)getOption ( "jsm.handler.property.type" );
```

```
    if ( type.equals ( "response" ) )
```

```
    {
```

```
        return true ;
```

```
    }
```

```
    return false ;
```

```
}
```

```

private final void setRequestMessage1 ( SOAPPart soapPart ) throws Except
{
    /*
        This needs to be the last message change

        Message.writeTo -> SOAPPart.writeTo methods will sent bytes as is

        This approach runs the risk of a FORM conversion taking place on the
    */

    if ( m_trace != null )
    {
        m_trace.println ( "Set request message 1 using byte[]" );
    }

    byte[] message = "any content sent as bytes".getBytes ( ENCODING_UT
    soapPart.setCurrentMessage ( message, SOAPPart.FORM_BYTES ) ;
}

private final void setRequestMessage2 ( MessageContext messageContext )
{
    /*
        With this example, I am using byte content from a file

        But you could have serialized a Document to a byte[] in-memory

        Or created a String and used that as the content argument

        etc..
    */

    if ( m_trace != null )
    {
        m_trace.println ( "Set request message 2 using byte[] message" );
    }

    File file = new File ( "ENVELOPE_REQUEST.XML" );

```

```

if ( m_trace != null )
{
    m_trace.println ( "MyHandler: set resquest message from external file :
}

// String content = "<?xml ...." ;

byte[] content = readFile ( file ) ;

Message requestMessage = new Message ( content ) ;

messageContext.setRequestMessage ( requestMessage ) ;
}

private final void setRequestMessage3 ( MessageContext messageContext, S
{
    if ( m_trace != null )
    {
        m_trace.println ( "Set request message 3 using document" ) ;
    }

    Document document = envelope.getAsDocument () ;

    /*
    Change document
    */

    byte[] content = serializeDocument ( document ) ;

    Message requestMessage = new Message ( content ) ;

    messageContext.setRequestMessage ( requestMessage ) ;
}

private final void setResponseMessage1 ( MessageContext messageContext
{
    /*
    With this example, I am using byte content from a file

```

But you could have serialized a Document to a byte[] in-memory

Or created a String and used that as the content argument

etc..

```
*/
```

```
File file = new File ( "ENVELOPE_RESPONSE.XML" );
```

```
if ( m_trace != null )
```

```
{
```

```
    m_trace.println ( "MyHandler: set response message from external file
```

```
}"
```

```
byte[] content = readFile ( file );
```

```
Message responseMessage = new Message ( content );
```

```
messageContext.setResponseMessage ( responseMessage );
```

```
}
```

```
private final byte[] serializeDocument ( Document document ) throws Except
```

```
{
```

```
    DOMImplementationRegistry registry = DOMImplementationRegistry.ne
```

```
    DOMImplementationLS implementation = (DOMImplementationLS)regi
```

```
    LSOutput output = implementation.createLSOutput ( );
```

```
    LSSerializer serializer = implementation.createLSSerializer ( );
```

```
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream ( 4
```

```
    output.setEncoding ( "utf-8" );
```

```
    output.setByteStream ( outputStream );
```

```
    serializer.setNewLine ( "\n" );
```

```

        serializer.write ( document, output ) ;

        return outputStream.toByteArray () ;
    }

private final void traceMessage ( Message message )
{
    if ( m_trace == null )
    {
        return ;
    }

    try
    {
        String fileName = "MYHANDLER_MESSAGE" + ServiceHelper.getS

        FileOutputStream outputStream = new FileOutputStream ( m_trace.cre

        message.writeTo ( outputStream ) ;

        outputStream.close () ;
    }
    catch ( Throwable t )
    {
        m_trace.print ( t ) ;
    }
}

private final void traceEnvelope ( SOAPEnvelope envelope )
{
    if ( m_trace == null )
    {
        return ;
    }

    /*
    This uses MessageElement - getAsDocument
        - getAsDocument

```

- getAsString

Converts element to String and parser into a Document

*/

try

{

Document document = envelope.getAsDocument () ;

DOMImplementationRegistry registry = DOMImplementationRegistry.

DOMImplementationLS implementation = (DOMImplementationLS)re

LSOutput output = implementation.createLSOutput () ;

LSSerializer serializer = implementation.createLSSerializer () ;

String fileName = "MYHANDLER_ENVELOPE" + ServiceHelper.get

FileOutputStream outputStream = new FileOutputStream (m_trace.cre

output.setEncoding ("utf-8") ;

output.setByteStream (outputStream) ;

serializer.setNewLine ("\n") ;

serializer.write (document, output) ;

outputStream.close () ;

}

catch (Throwable t)

{

m_trace.print (t) ;

}

}

private final void traceBody (SOAPEnvelope envelope) throws Exception

{

```

if ( m_trace == null )
{
    return ;
}

SOAPBody body = (SOAPBody)envelope.getBody () ;

if ( body == null )
{
    throw new IllegalArgumentException ( "no body available" ) ;
}

RPCElement operation = getOperation ( body ) ;

if ( operation == null )
{
    throw new IllegalArgumentException ( "no operation available" ) ;
}

m_trace.println ( "MyHandler: operation name : " + operation.getElemen

traceParameters ( operation ) ;
}

private final void traceParameters ( RPCElement operation ) throws Excepti
{
    if ( m_trace == null )
    {
        return ;
    }

    Vector vector = operation.getParams () ;

    int count = vector.size () ;

    m_trace.println ( "MyHandler: parameter count : " + count ) ;

    for ( int i=0; i < count; i++ )
    {

```

```

RPCParam parameter = (RPCParam)vector.get ( i ) ;

ParameterDesc parameterDesc = parameter.getParamDesc () ;

Class klazz = parameterDesc.getJavaType () ;

m_trace.println ( "MyHandler: parameter name : " + parameter.getElem

if ( klazz.isArray () )
{
    m_trace.println ( "MyHandler: parameter type : array of " + klazz.ge
}
else
{
    m_trace.println ( "MyHandler: parameter type : " + klazz.getName ()
}
}
}

private final RPCElement getOperation ( SOAPBody body )
{
    NodeList nodeList = body.getChildNodes () ;

    if ( nodeList == null )
    {
        return null ;
    }

    int count = nodeList.getLength () ;

    if ( count == 0 )
    {
        return null ;
    }

    for ( int i=0; i < count; i++ )
    {
        Node node = nodeList.item ( i ) ;

```

```

        if ( node instanceof RPCElement )
        {
            return(RPCElement)node ;
        }
    }

    return null ;
}

private final MessageElement getChildMessageElement ( MessageElement element )
{
    NodeList nodeList = element.getChildNodes () ;

    if ( nodeList == null )
    {
        return null ;
    }

    int count = nodeList.getLength () ;

    if ( count == 0 )
    {
        return null ;
    }

    for ( int i=0; i < count; i++ )
    {
        Node node = nodeList.item ( i ) ;

        if ( node instanceof RPCElement )
        {
            return(MessageElement)node ;
        }

        if ( node instanceof SOAPBodyElement )
        {
            return(MessageElement)node ;
        }
    }
}

```

```

    if ( node instanceof RPCParam )
    {
        return(MessageElement)node ;
    }

    if ( node instanceof MessageElement )
    {
        return(MessageElement)node ;
    }

    if ( node instanceof javax.xml.soap.SOAPElement )
    {
        /*
            Interface
        */
    }
}

return null ;
}

public final static byte[] readFile ( File file ) throws IOException
{
    int length = (int)file.length () ;

    byte[] content = new byte[length] ;

    FileInputStream inputStream = new FileInputStream ( file ) ;

    inputStream.read ( content ) ;

    inputStream.close () ;

    return content ;
}
}

```

2.18 Apache Axis 1.4 WS-Security and WS-Addressing

Refer to [2.17 SOAP Agent Message Handler](#).

Apache Axis 1.4 offers WS-Security capability by using the Apache WSS4J, XML-Security and WS-Addressing projects.

These projects do not support the latest changes in the implementation of WS-Security and WS-Addressing. Also these projects do not support new Web Services specifications such as WS-Policy, WS-Federation, WS-Trust and WS-SecureConversion.

2.19 REST Representational State Transfer

REST is an architectural style based on HTTP and XML or JSON technology. It is possible to develop REST style web services using JSMDirect and JSMPProxy.

Use the Apache HTTP Server URL rewrite capability to map JSMDirect and JSMPProxy URL's to REST resource style URL's.

REST Web Services Characteristics

- **Client-Server:** a pull-based interaction style.
- **Stateless:** each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server.
- **Cache:** to improve network efficiency responses must be capable of being labeled as cacheable or non-cacheable.
- **Uniform interface:** all resources are accessed with a generic interface (e.g., HTTP GET, POST, PUT, DELETE).
- **Named resources** - the system is comprised of resources that are named using a URL.
- **Interconnected resource representations** - the representations of the resources are interconnected using URLs, thereby enabling a client to progress from one state to another.
- **Layered components** - intermediaries, such as proxy servers, cache servers, gateways, etc, can be inserted between clients and resources to support performance, security, etc.

2.20 JCE Unlimited Strength Policy Files

Due to import control restrictions of some countries, the JCE jurisdiction policy files shipped with the Java SDK allow "strong" but limited cryptography to be used.

An "unlimited strength" version of these files indicating no restrictions on cryptographic strengths is available for those living in eligible countries (which is most countries).

You can download and replace the strong cryptography versions supplied with the Java SDK with the unlimited ones.

You need to update the two JAR files 'local_policy.jar' and 'US_export_policy.jar' files in the JDK's lib/security directory.

IBM's unlimited strength jurisdiction policy files:

www-128.ibm.com/developerworks/java/jdk/security/50

IBM's SDKs ship with strong but limited jurisdiction policy files. Unlimited jurisdiction policy files can be obtained from the link above. The ZIP file should be unpacked and the two JAR files placed in the JRE's jre/lib/security/ directory. These policy files are for use with IBM developed SDKs. The same files are used for the Version 1.4 and Version 5 SDKs.

The ZIP file should be unpacked and the two JAR files placed in the /QIBM/ProdData/Java400/jdk15/lib/security/ directory.

Oracle's unlimited strength jurisdiction policy files:

www.oracle.com/technetwork/java/javase/downloads/index.html

If these policy files are not installed then services that use Bouncy Castle will throw the following exception:

```
java.io.IOException: exception unwrapping private key -  
java.security.InvalidKeyException: Illegal key size
```

3. Java Service Manager Clients

3.1 LANSAR DML Client

3.2 LANSAR DMLX Client

3.3 ILE RPG Client

3.4 ILE RPGX Client

3.6 Tracing

3.7 Command

3.8 Data Areas

3.9 Resource properties - Studio sections

3.10 Resource properties - Internationalization

3.1 LANSAR DML Client

Warning. The RDML BIF has been changed to support 256 byte message and command parameters.

Any ILE program binding to the DCXS882X service program must still use 255 byte parameters.

Three Built-In Functions are required for a LANSAR DML client to have complete interaction with the Java Service Manager services.

These Built-In Functions only allow a single connection within the same job.

3.1.1 JSM_OPEN Open service.

3.1.2 JSM_COMMAND Send command.

3.1.3 JSM_CLOSE Close service.

The LANSAR developer only needs to understand how to use the commands offered by JSM services.

The overall structure of the LANSAR DML functions will be the same.

The function will perform the following operations:

- OPEN connection to the Java Service Manager.
- Issue a COMMAND to LOAD the service.
- Execute COMMANDs supported by the service.
- Issue a COMMAND to UNLOAD the service
- CLOSE connection to the Java Service Manager

Overview of field and list exchange

Command		No fields	
Command SERVICE_EXCHANGE(*FIELD *FIELDS)		Fields	
Command	#WRKLST	Fields	
Command SERVICE_EXCHANGE(*FIELD *FIELDS)	#WRKLST	Fields	

Note 1: All function fields are sent with a list.

Note 2: Command keyword SERVICE_LIST(...) is required for working list #WRKLST.

3.1.1 JSM_OPEN

The JSM_OPEN Built-In Function is always executed first. It is used to connect the JSM client to the Java Service Manager and to start a thread for the service. Each client will have its own thread.

Syntax:

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Optional	Server	50	50		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Status	20	20		
2	A	Required	Message	1	256		

The server argument is optional for the connection. If the server argument is not supplied, or if the argument value contains blanks, then the remote host and port number is obtained from the data area JSMCLTDTA on IBM i or from file jsmcltdta.txt on Windows or Linux.

The server argument can have any of the following values:

host:port	Use specified host and port
host	Use host and get port from JSMCLTDTA
:port	Use port and get host from JSMCLTDTA

:	Get host and port from JSMCLTDTA
	Get host and port from JSMCLTDTA

The JSM_OPEN must be executed before the JSM_COMMAND can be used. The JSM_OPEN Built-In Function performs a DNS lookup to resolve the host name address. Using a dotted decimal IP address does not get around this DNS lookup.

If the JSM_OPEN Built-In Function is slow to open a connection then you could have a DNS lookup issue. Add an entry to the local host table to improve performance.

TCP/IP connection performance can be improved if the registered DNS servers are available and performing well.

Change TCP/IP Domain

```
Host name search priority      *LOCAL
Domain name server
Internet address              139.130.4.4
                              203.48.48.13
```

If the first Domain Name Server (DNS) in the list does not respond, the second DNS server in the list will be contacted. If the second DNS server does not respond, the third DNS server will be contacted.

See [IBM i - Network Performance](#) for further information.

3.1.2 JSM_COMMAND

The JSM_COMMAND Built-In Function is used to load and unload services and execute commands supported by the service. The JSM_COMMAND can only be executed once a JSM_OPEN has completed successfully.

Syntax:

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Command	1	256		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Status	20	20		
2	A	Required	Message	1	256		
3	L	Optional	Working List				

A service is loaded before commands to the service are executed. A service is unloaded once you are finished using it. Only one service can be loaded at a time. Refer to [3.7 Command](#).

The JSM_COMMAND has a number of:

[RDML Reserved Commands](#)

[RDML Reserved Keywords](#)

[RDML Reserved Field Names](#)

For command services details, refer to [Java Service Manager Services](#).

RDML Reserved Commands

Any command or keyword that starts with SERVICE_ is reserved.

For example, these commands are reserved:

SERVICE_LOAD	SERVICE(servicename) TRACE(option) TRACE_
SERVICE_SET	TRACE(*CLEAR)
SERVICE_GET	PROPERTY(property)
SERVICE_RECLAIM	
SERVICE_UNLOAD	

The SERVICE_LOAD command instructs the JSM service thread to load and instantiate the specified service program. Only one service class is loaded at a time. The SERVICE_LOAD command is a good place for the Java service programmer to write the code to load default values being used with the current service.

The SERVICE_SET command allows the enabling of trace file clearing.

The SERVICE_GET command allows access to the keyword/value properties from the associated service properties file. To reduce the conflict between user-defined keyword names and current and future LANSAs names prefix the keyword with a global unique name. The service property keyword is case insensitive.

The service property keyword value is returned in the #JSMMSG field.

If the keyword does not exist, then a status of NOT_EXIST is returned.

The special PROPERTY values *SERVICE and *SERVICECLASS return the loaded service class name.

Example

```
com.acme.property.messageType=html
#
message.001=Service successfully loaded
message.002=Service successfully unloaded
message.003=Command is not supported :
```

The SERVICE_RECLAIM command allows an explicit garbage collection to be done.

You could use this command in conjunction with the JVM `-Xcompactexplicitgc` option.

The `SERVICE_UNLOAD` command instructs the JSM service thread to invalidate the currently loaded service program and expect to load another service. This command is optional and does not need to be called if the JSM connection is going to be closed. (The `SERVICE_UNLOAD` command is a good place for the Java service programmer to put code to release currently allocated resources.)

RDML Reserved Keywords

The following keywords are reserved:

SERVICE_LIST

SERVICE_EXCHANGE

TRIM

TRUNCATE

When using a working list argument with the JSM_COMMAND BIF, a SERVICE_LIST keyword must be included with the command.

The SERVICE_LIST keyword is used to describe the list argument being passed to the JSM command as shown in the following example:

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEF_LIST NAME(#WRKLIST) FIELDS(#DEPARTMENT,#DEPTDESC) TYPI
USE BUILTIN(JSM_COMMAND) WITH_ARGS('RECEIVE HANDLER(IX
```

Note: The SERVICE_LIST keyword fields do not require the # prefix.

If the list of fields is very long, it may exceed the size limit of a JSM command string. In this case, externalize the list of fields to an entry in the list.properties file located in the instance system sub-directory as shown in the following example:

```
list.js016.receive01=DEPARTMENT,DEPTDESC
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('RECEIVE HANDLER(IX
```

If the SERVICE_LIST keyword only contains one entry then the command parser will check if this single value is a program field. If this value is not a program field, then it is used as a lookup value on the list.properties file to locate a field list entry. It is recommended that you use a naming convention of function name plus dot plus command + sequence (JS016.RECEIVE01). This allows for an easy identification of what programs are using a particular entry when the list.properties file is viewed.

Note: An implied SERVICE_EXCHANGE(*FIELD) is done when the SERVICE_LIST keyword is used.

The SERVICE_EXCHANGE keyword is used to trigger the exchange between the LANSAs function field values and the currently loaded JSM service. The only values supported by this keyword is *FIELD or *FIELDS.

The LANSAs BIF performs a scan on the command string and searches for the following pattern:

SERVICE_EXCHANGE(*FIELD)

SERVICE_EXCHANGE(*FIELDS)

The keyword and its value must be identical to the examples above and without imbedded spaces.

Note: If a working list and the associated SERVICE_LIST keyword is used, then all fields are passed and a SERVICE_EXCHANGE(*FIELD) or SERVICE_EXCHANGE(*FIELDS) is not required on the same command.

RDML Reserved Field Names

Any fields that start with the prefix value contained in the JSMCLTDTA data area will be excluded from the field list transfer. This prefix value occupies positions 51 to 60 of the data area JSMCLTDTA. For Windows or Linux, this prefix value can be set with a keyword value pair in file jsmcltdta.txt. The keyword to specify the prefix is ExcludePrefix (Example: ExcludePrefix=JSM). The default value for this prefix is JSM. This prefix allows the JSMSTS, JSMMSG and JSMCMD variables to be excluded from the field list transfer. For more information refer to [3.8 Data Areas](#).

3.1.3 JSM_CLOSE

The JSM_CLOSE Built-In Function is used to end the connection to the Java Service Manager. It is good programming practice to use a JSM_CLOSE before ending your function.

The SERVICE_UNLOAD does not have to be executed if a JSM_CLOSE is being executed.

If your function is using JSMDirect, do not use the SERVICE_UNLOAD command just issue the JSM_CLOSE to send the last command byte array response back to the HTTP client.

Syntax:

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
----	------	----------------------	-------------	---------------	---------------	----------------	-------------

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Status	20	20		
2	A	Required	Message	1	256		

3.1.4 Sample LANSAs RDML Client Programs

Following is a very simple LANSAs function that opens and then closes a connection to the JSM. If an error occurs, the status and message are displayed. This program does not use any services.

For further examples of RDML client applications, refer to [Client Application Examples](#).

Example 1 - Using default (JSMCLTDTA data area)

```
FUNCTION OPTIONS(*DIRECT)
*
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
*
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
IF COND(#JSMSTS *NE OK')
DISPLAY FIELDS(#JSMSTS #JSMMSG)
ENDIF
*
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
IF COND(#JSMSTS *NE OK')
DISPLAY FIELDS(#JSMSTS #JSMMSG)
ENDIF
```

Example 2 - Using server argument

```
FUNCTION OPTIONS(*DIRECT)
*
DEFINE FIELD(#JMSRV) TYPE(*CHAR) LENGTH(50)
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
*
CHANGE FIELD(#JMSRV) TO('LOCALHOST:4560')
*
USE BUILTIN(JSM_OPEN) WITH_ARGS(#JMSRV)TO_GET(#JSMSTS #
```

```

IF COND(#JSMSTS *NE OK')
DISPLAY FIELDS(#JSMSTS #JSMMSG)
ENDIF
*
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
IF COND(#JSMSTS *NE OK')
DISPLAY FIELDS(#JSMSTS #JSMMSG)
ENDIF

```

The following example shows how the JSM BIFs are used together and in what order. This is not a complete RDML function. In this example JSM_OPEN opens a connection to the Java Service Manager and starts a thread.

A series of JSM_COMMANDs are issued to:

- Load the FTPService service
- Connect to the host
- Login using the specified user/password
- Change the directory path
- Set mode to binary
- Put the file to the FTP site
- Quit the FTP session
- Unload the service.

JSM_CLOSE ends the connection.

```

FUNCTION OPTIONS(*DIRECT)

```

```

DEFINE FIELD(#JMSRV) TYPE(*CHAR) LENGTH(50)
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)

```

```

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)

```

```

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVICE_LOAD')

```

```

USE BUILTIN(JSM_COMMAND) WITH_ARGS('CONNECT HOST(LOCALHOST)')

```

USE BUILTIN(JSM_COMMAND) WITH_ARGS('LOGIN USER(user) PASS
USE BUILTIN(JSM_COMMAND) WITH_ARGS('CHGDIR PATH(/JSM)') T
USE BUILTIN(JSM_COMMAND) WITH_ARGS('BINARY') TO_GET(#JSM
USE BUILTIN(JSM_COMMAND) WITH_ARGS('PUT FROM(file) TO(/TM
USE BUILTIN(JSM_COMMAND) WITH_ARGS('QUIT') TO_GET(#JSMST
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TC
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)

3.2 LANSAR DMLX Client

Three Built-In Functions are required for a LANSAR DMLX client to have complete interaction with the Java Service Manager services.

These Built-In Functions allow multiple concurrent connections in the same function.

- 3.2.1 JSMX_BEGIN Initialize or re-initialize the service program internal state.
- 3.2.2 JSMX_OPEN Open service
- 3.2.3 JSMX_COMMAND Send command.
- 3.2.4 JSMX_CLOSE Close service.
- 3.2.5 JSMX_END End and re-initialize the service program internal state.

The LANSAR developer only needs to understand how to use the commands offered by the JSM services.

The overall structure of the LANSAR DMLX functions will be the same.

The function will perform the following operations:

- BEGIN using the Built-In Functions.
- OPEN connection to the Java Service Manager.
- Issue a COMMAND to LOAD the service.
- Execute COMMANDS supported by the service.
- Issue a COMMAND to UNLOAD the service
- CLOSE connection to the Java Service Manager
- END using the Built-In Functions.

Overview of field and list exchange

Command		

Command SERVICE_EXCHANGE(*FIELD *FIELDS)		
Command		#WRKLS1
Command	#FLDLST	
Command	#FLDLST	#WRKLS1
Command SERVICE_EXCHANGE(*FIELD *FIELDS)		#WRKLS1
Command SERVICE_EXCHANGE(*FIELD *FIELDS)	#FLDLST	#WRKLS1

Note: Specified fields takes precedence over SERVICE_EXCHANGE(*FIELD) or SERVICE_EXCHANGE(*FIELDS) keyword.

3.2.1 JSMX_BEGIN

The JSMX_BEGIN Built-In Function must be the first API call in a logical unit of processing.

This initializes the internal state of the service program by closing any open connections and freeing any allocated resources. The internal state of the service program is scoped to the job.

3.2.2 JSMX_OPEN

The JSMX_OPEN Built-In Function is always executed first. It is used to connect the JSMX client to the Java Service Manager and to start a thread for the service. Each client will have its own thread.

Syntax:

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Optional	Server	1	50		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Ma De
1	A	Required	Status	1	20		
2	A	Required	Message	1	Unlimited		
3	A	Required	Connection Handle	4	4		

The server argument is optional for the connection. If the server argument is not supplied, or if the argument value contains blanks, then the remote host and port number is obtained from the data area JSMCLDTA on IBM i or from file jsmcltda.txt on Windows or Linux.

The server argument can have any of the following values:

host:port	Use specified host and port

host	Use host and get port from JSMCLTDTA
:port	Use port and get host from JSMCLTDTA
:	Get host and port from JSMCLTDTA
	Get host and port from JSMCLTDTA

The JSMX_OPEN must be executed before the JSMX_COMMAND can be used.

The JSMX_OPEN Built-In Function performs a DNS lookup to resolve the host name address. Using a dotted decimal IP address does not get around this DNS lookup.

If the JSMX_OPEN Built-In Function is slow to open a connection then you could have a DNS lookup issue. Add an entry to the local host table to improve performance.

TCP/IP connection performance can be improved if the registered DNS servers are available and performing well.

Change TCP/IP Domain

```
Host name search priority      *LOCAL
Domain name server
Internet address              139.130.4.4
                              203.48.48.13
```

If the first Domain Name Server (DNS) in the list does not respond, the second DNS server in the list will be contacted. If the second DNS server does not respond, the third DNS server will be contacted.

See [IBM i - Network Performance](#) for further information.

3.2.3 JSMX_COMMAND

The JSMX_COMMAND Built-In Function is used to load and unload services and execute commands supported by the service. The JSMX_COMMAND can only be executed once a JSMX_OPEN has completed successfully.

If an optional working list argument is specified then the fields defined in that list are available to the loaded service. If no working list argument is specified then no fields are available to the loaded service. This field list does not require an entry only the list definition is used to determine which fields are sent to the JSM service.

If an optional working list return value is specified then that working list is available to the loaded service.

Syntax:

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Ma De
1	A	Required	Connection Handle	4	4		
2	A	Required	Command	1	Unlimited		
3	L	Optional	Field List				

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Ma De
1	A	Required	Status	1	20		
2	A	Required	Message	1	Unlimited		
3	L	Optional	Working List				

A service is loaded before commands to the service are executed. A service is unloaded once you are finished using it. Only one service can be loaded at a time. Refer to [3.7 Command](#).

The JSMX_COMMAND has a number of:

[RDMLX Reserved Commands](#)

[RDMLX Reserved Keywords](#)

[RDMLX Reserved Field Names](#)

For command services details, refer to [Java Service Manager Services](#).

Technical Note 1

For a service to receive HTTP posted content, the SERVICE_CONTENT(*HTTP) keyword is required on the SERVICE_LOAD command.

If this is the first connection to use the SERVICE_CONTENT(*HTTP) keyword, then the posted HTTP content is read and sent to the service.

This connection takes responsibility for sending the HTTP response.

If this is NOT the first connection to use the SERVICE_CONTENT(*HTTP) keyword, then this connection only takes responsibility for sending the HTTP response. The SERVICE_LOAD command does NOT receive the HTTP content, but does receive the HTTP keywords.

When the connection that is responsible for the HTTP response uses the JSMX_CLOSE to close the connection, the returned byte array response becomes the HTTP response.

Only one connection can have responsibility for sending the HTTP response at a time.

HTTP content can only be read once.

HTTP response can only be written once.

HTTP keywords are always sent with the SERVICE_LOAD command.

Scenario A

#1 JSMX_OPEN - open connection

#2 JSMX_OPEN - open connection

#3 JSMX_OPEN - open connection

#1 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)
SERVICE_CONTENT(*HTTP)") - send CGI keywords, read STDIN, claim
ownership
#2 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)") - send CGI
keywords
#3 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)
SERVICE_CONTENT(*HTTP)") - send CGI keywords, transfer ownership
#1 JSMX_CLOSE - close connection
#2 JSMX_CLOSE - close connection
#3 JSMX_CLOSE - close connection and write STDOUT

Scenario B

#1 JSMX_OPEN - open connection
#1 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)
SERVICE_CONTENT(*HTTP)") - send CGI keywords, read STDIN, claim
ownership
#1 JSMX_CLOSE - close connection and write STDOUT
#2 JSMX_OPEN - open connection
#2 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)") - send CGI
keywords
#2 JSMX_CLOSE - close connection
#3 JSMX_OPEN - open connection
#3 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)
SERVICE_CONTENT(*HTTP)") - send CGI keywords, transfer ownership
#3 JSMX_CLOSE - close connection

Technical Note 2

The following data types are supported. If any other data type is used, that field will be ignored and not passed to the JSM service.

TYPE(*CHAR)
TYPE(*NCHAR)
TYPE(*STRING)
TYPE(*INT)
TYPE(*DEC)
TYPE(*FLOAT)
TYPE(*PACKED)

TYPE(*SIGNED)
TYPE(*BOOLEAN)
TYPE(*DATE)
TYPE(*TIME)
TYPE(*DATETIME)
TYPE(*BLOB)
TYPE(*CLOB)

RDMLX Reserved Commands

Any command or keyword that starts with SERVICE_ is reserved.

For example, these commands are reserved:

SERVICE_LOAD	SERVICE(servicename) TRACE(option) TRACE_
SERVICE_SET	TRACE(*CLEAR)
SERVICE_GET	PROPERTY(property)
SERVICE_RECLAIM	
SERVICE_UNLOAD	

The SERVICE_LOAD command instructs the JSM service thread to load and instantiate the specified service program. Only one service class is loaded at a time. The SERVICE_LOAD command is a good place for the Java service programmer to write the code to load default values being used with the current service.

The SERVICE_SET command allows the enabling of trace file clearing.

The SERVICE_GET command allows access to the keyword/value properties from the associated service properties file. To reduce the conflict between user-defined keyword names and current and future LANSAs names prefix the keyword with a global unique name. The service property keyword is case insensitive.

The service property keyword value is returned in the #JSMMSG field.

If the keyword does not exist, then a status of NOT_EXIST is returned.

The special PROPERTY values *SERVICE and *SERVICECLASS return the loaded service class name.

Example

```
com.acme.property.messageType=html
#
message.001=Service successfully loaded
message.002=Service successfully unloaded
message.003=Command is not supported :
```

The SERVICE_RECLAIM command allows an explicit garbage collection to be done.

You could use this command in conjunction with the JVM `-Xcompactexplicitgc` option.

The `SERVICE_UNLOAD` command instructs the JSM service thread to invalidate the currently loaded service program and expect to load another service. This command is optional and does not need to be called if the JSM connection is going to be closed. (The `SERVICE_UNLOAD` command is a good place for the Java service programmer to put code to release currently allocated resources.)

RDMLX Reserved Keywords

The following keywords are reserved:

SERVICE_CONTENT

SERVICE_EXCHANGE

TRIM

TRUNCATE

The SERVICE_CONTENT keyword is used on the SERVICE_LOAD command to trigger the exchange of content from the LANSAs function to the service being loaded. The only value supported by this keyword is *HTTP.

The LANSAs BIF performs a scan on the SERVICE_LOAD command string and searches for the following pattern:

SERVICE_CONTENT(*HTTP)

The keyword and its value must be identical to the example above, entirely in uppercase and without imbedded spaces.

The SERVICE_EXCHANGE keyword is used to trigger the exchange between the LANSAs function field values and the currently loaded JSM service. The only values supported by this keyword is *FIELD or *FIELDS. The *FIELDS value will include BLOB and CLOB fields into the list of exchanged fields.

The LANSAs BIF performs a scan on the command string and searches for the following pattern:

SERVICE_EXCHANGE(*FIELD)

SERVICE_EXCHANGE(*FIELDS)

The keyword and its value must be identical to the example above and without imbedded spaces.

RDMLX Reserved Field Names

Any fields that start with the prefix value contained in the JSMCLTDTA data area will be excluded from the field list transfer. This prefix value occupies positions 51 to 60 of the data area JSMCLTDTA. For Windows or Linux, this prefix value can be set with a keyword value pair in file jsmcltdta.txt. The keyword to specify the prefix is ExcludePrefix (Example: ExcludePrefix=JSM). The default value for this prefix is JSM. This prefix allows the JSMSTS, JSMMSG and JSMCMD variables to be excluded from the field list transfer. For more information refer to [3.8 Data Areas](#).

3.2.4 JSMX_CLOSE

The JSMX_CLOSE Built-In Function is used to end the connection to the Java Service Manager. It is good programming practice to use a JSMX_CLOSE before ending your function.

The SERVICE_UNLOAD does not have to be executed if a JSMX_CLOSE is being executed.

If your function is using JSMDirect, do not use the SERVICE_UNLOAD command just issue the JSM_CLOSE to send the last command byte array response back to the HTTP client.

Syntax:

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Decim
1	A	Required	Connection Handle	4	4		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Ma De
1	A	Required	Status	1	20		
2	A	Required	Message	1	Unlimited		

3.2.5 JSMX_END

The JSMX_END Built-In Function is the last API call in a logical unit of processing.

This initializes/finalizes the internal state of the service program by closing any open connections and freeing any allocated resources. The internal state of the service program is scoped to the job.

You should not call this Built-In Function while any JSM connections remain open if you wish to continue to use the open connection(s).

3.2.6 Sample LANSAR DMLX Client Programs

For examples of RDMLX client applications, refer to [Client Application Examples](#).

3.3 ILE RPG Client

WARNING. The RDML BIF has been changed to support 256 byte message and command parameters.
Any ILE RPG program binding to the DCXS882X service program must still use 255 byte parameters.

Four API calls are required for an ILE RPG client to have complete interaction with the Java Service Manager services.

These APIs only allow a single connection within the same job.

3.3.1 JSOPEN Open service.

3.3.2 JSMCMD & JSMCMDX Send command.

3.3.3 JSMCLOSE Close service.

The RPG developer only needs to understand how to use the commands offered by the JSM services.

The overall structure of the RPG programs will be the same.

The program will perform the following operations:

- OPEN connection to the Java Service Manager.
- Issue a COMMAND to LOAD the service.
- Execute COMMANDs supported by the service.
- Issue a COMMAND to UNLOAD the service
- CLOSE connection to the Java Service Manager

On IBM i the ILE RPG client program needs to be bound to service program DCXS882X and this service program needs to be shipped with the client program. The client program is also dependent on data area JSMCLTDTA and JSMMSGF message file.

Example source code and how to create a program are located in files QRPGLSRC and QCLSRC in the JSM library (as nominated during the LANSAs Integrator install).

Overview of field and list exchange

Command	No fields	No list
Command SERVICE_STRUCTURE(...)	Fields	No

		list
Command SERVICE_STRUCTURE(...) OCCURS(...) SIZE(...) COUNT(...)	No fields	List

Note: You cannot have both fields and a list.

3.3.1 JSMOPEN

The JSMOPEN is always executed first. It is used to connect the JSM client to the Java Service Manager and to start a thread for the service. Each client will have its own thread.

The server argument is optional for the connection. If the server argument is not supplied, or if the argument value contains blanks, then the remote host and port number is obtained from the data area JSMCLTDTA on IBM i.

The server argument can have any of the following values:

host:port	Use specified host and port
host	Use host and get port from JSMCLTDTA
:port	Use port and get host from JSMCLTDTA
:	Get host and port from JSMCLTDTA
	Get host and port from JSMCLTDTA

The JSMOPEN must be executed before the JSMCMD can be used.

The JSMOPEN API performs a DNS lookup to resolve the host name address. Using a dotted decimal IP address does not get around this DNS lookup.

If the JSMOPEN API is slow to open a connection then you could have a DNS lookup issue. Add an entry to the local host table to improve performance.

TCP/IP connection performance can be improved if the registered DNS servers are available and performing well.

Change TCP/IP Domain	
Host name search priority	*LOCAL
Domain name server	
Internet address	139.130.4.4
	203.48.48.13

If the first Domain Name Server (DNS) in the list does not respond, the second DNS server in the list will be contacted. If the second DNS server does not respond, the third DNS server will be contacted.

See [IBM i - Network Performance](#) for further information.

3.3.2 JSMCMD & JSMCMDX

The JSMCMD or JSMCMDX are used to load and unload services and to execute commands supported by the service. The JSMCMD or JSMCMDX can only be executed once a JSOPEN has completed successfully.

JSMCMDX is only available to ILE clients and allows the contents of a variable to be sent with the command string. The service programmer accesses a copy of this variable byte array (sent by JSMCMDX) by using the JSMCMD `getByteArray()` method.

Example

```
byte[] clientData = command.getByteArray() ;
```

Two API's are used to retrieve a byte array returned by the JSM service. Depending on the service, this array may also contain fields or working lists. JSMBYTELENGTH places the length of the last returned byte array into byte length parameter. JSMBYTERECV copies the contents of the byte array into the variable parameter.

Note that the byte array parameter must be big enough to hold the received bytes.

```
C  CALLB  'JSMBYTERECV'  
C  PARM   Details
```

The command string sent by the JSMCMDX API is converted from the client encoding to Unicode.

The byte array is not converted and is passed to the service unprocessed.

To send single or multiple occurrence data structures using RPG, refer to [3.3.6 RPG Data Structure](#).

A service is loaded before commands to the service are executed. A service is unloaded once you are finished using it. Only one service can be loaded at a time. Refer to [3.7 Command](#).

The JSMCMDX or JSMCMDX API has a number of:

[3.3.4 RPG Reserved Commands](#)

[3.3.5 RPG Reserved Keywords](#)

For command services details, refer to [Java Service Manager Services](#).

3.3.3 JSMCLOSE

The JSMCLOSE is used to end the connection to the Java Service Manager. It is good programming practice to use a JSMCLOSE before ending your function.

The SERVICE_UNLOAD does not have to be executed if a JSMCLOSE is being executed.

If your function is using JSMDirect, do not use the SERVICE_UNLOAD command, just issue the JSMCLOSE to send the last command byte array response back to the HTTP client.

3.3.4 RPG Reserved Commands

Any command or keyword that starts with SERVICE_ is reserved.

For example, these commands are reserved:

SERVICE_LOAD	SERVICE(servicename) TRACE(option) TRACE_
SERVICE_SET	TRACE(*CLEAR)
SERVICE_GET	PROPERTY(property)
SERVICE_RECLAIM	
SERVICE_UNLOAD	

The SERVICE_LOAD command instructs the JSM service thread to load and instantiate the specified service program. Only one service class is loaded at a time. The SERVICE_LOAD command is a good place for the Java service programmer to write the code to load default values being used with the current service.

The SERVICE_SET command allows the enabling of trace file clearing.

The SERVICE_GET command allows access to the keyword/value properties from the associated service properties file. To reduce the conflict between user-defined keyword names and current and future LANSAs names prefix the keyword with a global unique name. The service property keyword is case insensitive.

The service property keyword value is returned in the #JSMMSG field.

If the keyword does not exist, then a status of NOT_EXIST is returned.

The special PROPERTY values *SERVICE and *SERVICECLASS return the loaded service class name.

Example

```
com.acme.property.messageType=html
#
message.001=Service successfully loaded
message.002=Service successfully unloaded
message.003=Command is not supported :
```

The SERVICE_RECLAIM command allows an explicit garbage collection to be done.

You could use this command in conjunction with the JVM `-Xcompactexplicitgc` option.

The `SERVICE_UNLOAD` command instructs the JSM service thread to invalidate the currently loaded service program and expect to load another service. This command is optional and does not need to be called if the JSM connection is going to be closed. (The `SERVICE_UNLOAD` command is a good place for the Java service programmer to put code to release currently allocated resources.)

3.3.5 RPG Reserved Keywords

The following keywords are reserved:

SIZE

COUNT

OCCURS

SERVICE_STRUCTURE

TRIM

TRUNCATE

3.3.6 RPG Data Structure

RPG data structure support allows RPG programs to send and receive the single or multiple occurrence data structures between the RPG program and the current JSM service.

RPG programs can pass data structures to the JSM services these data structures appear as list or field list objects.

The RPG program can only pass a field list object or a list object with each command, a LANSAs function can pass both a field list object and list object at the same time.

The JSMCMDX call is used to pass the data structure as a byte array.

If the command keyword SERVICE_STRUCTURE is present the byte array is converted into a JSMList or JSMFieldList object depending on other command keywords.

These objects are available from the JSMCMD object.

```
JSMList list = command.getList () ;
```

```
JSMFieldList fieldList = command.getFieldList () ;
```

If no SERVICE_STRUCTURE keyword is present then the data structure can be accessed by the getByteArray method.

```
byte[] data = command.getByteArray () ;
```

Reserved keywords

- SERVICE_STRUCTURE (xxx) - mandatory if a structure needs to be used.
- OCCURS (nnn) - mandatory for list
- SIZE (nnn) - mandatory for list
- COUNT (nnn) – optional for list, specifies the number of valid entries in the list.

Data structure as a list

To send a multiple occurrence data structure across to the JSM service as a list object, the programmer needs to include the keyword OCCURS.

- OCCURS keyword specifies the maximum number of entries in the list.
- COUNT keyword specifies the current number of entries in the list. Default is zero.
- SIZE keyword is required when an OCCURS keyword is used. This is used as a check against the calculated entry size.
- The byte array length must be equal to SIZE value times the OCCURS value.

Data structure as a field list

To send a data structure across to the JSM service as a field list object the programmer does not need to use the COUNT, OCCURS or SIZE keywords.

- COUNT keyword value defaults to one.
- OCCURS keyword value defaults to one.
- The byte array length must be equal to the calculated structure size.

Determining the layout of the structure

The value of the SERVICE_STRUCTURE keyword is used as a keyed lookup of the structure.properties file in the system sub-directory.

The value component of this property entry is the location of the XML file defining the structure layout.

```
#
# Java Service Manager structures
#
structure.demoxml.orderhead=structure/demoxml-orderhead.xml
structure.demoxml.orderline=structure/demoxml-orderline.xml
```

Example structure XML

```
<?xml version="1.0" encoding="UTF-8"?>

<rdml:structure xmlns:rdml="http://www.lansa.com/2000/XML/Function">

  <rdml:field name="ORDER" type="S" length="10" />
  <rdml:field name="NAME" type="A" length="50" />
  <rdml:field name="STREET" type="A" length="50" />
  <rdml:field name="CITY" type="A" length="50" />
```

```

    <rdml:field name="STATE" type="A" length="5" />
    <rdml:field name="ZIP" type="A" length="5" />

</rdml:structure>

```

Example RPG program

```

* Order head
D ORDERHEAD      DS
D ORDER      10S 0  INZ(0)
D NAME       50    INZ(' ')
D STREET     50    INZ(' ')
D CITY       50    INZ(' ')
D STATE      5     INZ(' ')
D ZIP        5     INZ(' ')
D HEADSIZE   C     %SIZE(ORDERHEAD)

```

```

* Order lines, up to 10 lines
D ORDERLINE DS OCCURS(10)
D LINENUM   3S 0  INZ(0)
D PARTNUM   3S 0  INZ(0)
D PARTDSC   50    INZ(' ')
D PARTAMT   10P 2 INZ(0)
D PARTQTY   3P 0  INZ(0)
D LINEELEM  C     %ELEM(ORDERLINE)
D LINESIZE  C     %SIZE(ORDERLINE)

```

This command will receive the single field values into the ORDERHEAD structure:

```

RECEIVE HANDLER(IXML) XSL(RECEIVEORDER)
SERVICE_STRUCTURE(DEMOXML.ORDERHEAD)

```

This command will receive the multiple order lines into the ORDERLINE structure:

```

RECEIVE HANDLER(IXML) XSL(RECEIVEORDER)
SERVICE_STRUCTURE(DEMOXML.ORDERLINE) SIZE(64)
OCCURS(10)

```

The data structure names do not need to match the names in the structure XML

file, it is only the data type and position that are used.

The structure XML names need to match the field names in the XSL file.

The shipped RPG example QRPGLSRC/DEMOXML illustrates how to send and receive data structures.

3.3.7 Sample ILE RPG Client Programs

For examples of ILE RPG client applications, refer to [Client Application Examples](#).

3.4 ILE RPGX Client

Seven API calls are required for an ILE RPG client to have complete interaction with the Java Service Manager services.

These APIs allow multiple connections within the same job.

3.4.1 JSMX_BEGIN	Initialize or re-initialize the service program internal state.
3.4.2 JSMX_OPEN	Open service.
3.4.3 JSMX_BINDFLD	Bind program fields to connection handle.
3.4.4 JSMX_BINDLST	Bind program list to connection handle.
3.4.5 JSMX_COMMAND	Send command.
3.4.6 JSMX_CLOSE	Close service.
3.4.7 JSMX_END	End and re-initialize the service program internal state.

The RPG developer only needs to understand how to use the commands offered by the JSM services.

The overall structure of the RPG programs will be the same.

The program will perform the following operations:

- BEGIN using the API interface.
- OPEN connection to the Java Service Manager.
- Issue a COMMAND to LOAD the service.
- Execute COMMANDs supported by the service.
- Issue a COMMAND to UNLOAD the service
- CLOSE connection to the Java Service Manager
- END using the API interface.

On IBM i the ILE RPG client program needs to be bound to service program JSMRPGSRV and this service program needs to be shipped with the client program. The client program is also dependent on data area JSMCLDTA and JSMMSGF message file.

Example source code and how to create a program are located in files QRPGLSRC and QCLSRC in the JSM library (as nominated during the LANSa Integrator install).

Most of the service program API character string parameters can be of any size, as the service program will determine the length of character input using IBM's CEEGSI API. The connection handle parameter is always character of length 4.

Overview of field and list exchange

Command	No fields	No list
Command + JSMX_BINDFLD	Fields	No list
Command + JSMX_BINDLST	No fields	List
Command + JSMX_BINDFLD + JSMX_BINDLST	Fields	List

3.4.1 JSMX_BEGIN

The JSMX_BEGIN API must be the first API call in a logical unit of processing.

This initializes the internal state of the service program by closing any open connections and freeing any allocated resources. The internal state of the service program is scoped to the job.

Parameters

Number	Symbolic Name	Description
1	JSMXCFG	Input, structure. Usage not yet defined.
2	JSMXLEN	Input, integer 10. Length of JSMXCFG.

3.4.2 JSMX_OPEN

The JSMX_OPEN API is used to open a new connection from the client application to the Java Service Manager. The client may have more than one (up to 100) connections open simultaneously by making multiple calls to JSMX_OPEN.

Parameters

Number	Symbolic Name	Description
1	JSMHDL	Output, character 4. The JSMX_OPEN call returns the connection handle it assigns to this connection. This handle is used on subsequent JSMX_COMMAND and other API calls to identify the connection to which the call applies.
2	JMSRV	Input, character (variable, recommended 50). Identifies the server to which the client application wishes to open a connection.
3	JSMSTS	Output, character (variable, recommended 20). The JSM server returns the status of the operation in this field. The client application can test this field to determine whether the operation succeeded. In most cases a status of 'OK' indicates successful completion.
4	JSMMSG	Output, character (variable, recommended 512). The JSM server may return a message concerning the operation in this field - for example a completion message or an error message when the operation fails.

The server argument is optional for the connection. If the server argument is not supplied, or if the argument value contains blanks, then the remote host and port

number is obtained from the data area JSMCLTDTA on IBM i.
The server argument can have any of the following values:

host:port	Use specified host and port
host	Use host and get port from JSMCLTDTA
:port	Use port and get host from JSMCLTDTA
:	Get host and port from JSMCLTDTA
	Get host and port from JSMCLTDTA

The JSMX_OPEN must be executed before the JSMX_COMMAND can be used.

The JSMX_OPEN Built-In Function performs a DNS lookup to resolve the host name address. Using a dotted decimal IP address does not get around this DNS lookup.

If the JSMX_OPEN Built-In Function is slow to open a connection then you could have a DNS lookup issue. Add an entry to the local host table to improve performance.

TCP/IP connection performance can be improved if the registered DNS servers are available and performing well.

```
Change TCP/IP Domain
Host name search priority      *LOCAL
Domain name server
Internet address               139.130.4.4
                               203.48.48.13
```

If the first Domain Name Server (DNS) in the list does not respond, the second DNS server in the list will be contacted. If the second DNS server does not respond, the third DNS server will be contacted.

See [IBM i - Network Performance](#) for further information.

3.4.3 JSMX_BINDFLD

The JSMX_BINDFLD API is optional and is used to "bind" fields to the specified connection handle so that the field values can be sent or received by a subsequent JSMX_COMMAND API call.

Only one set of fields may be bound at once – therefore you should include all required fields and their descriptions in a single JSMX_BINDFLD call before the JSMX_COMMAND call(s) in which they might be used.

The JSMX_BINDFLD call is not persistent – that is, it lasts only until the next JSMX_COMMAND call completes.

You must call JSMX_BINDFLD again before a subsequent JSMX_COMMAND API call, if the service command requires bound fields. (The same is true for bound lists – see the JSMX_BINDLST.)

To "unbind" fields from the specified connection handle, you can pass a null value (use *OMIT in ILE RPG) for the JSMMETA or JSMDATA parameters or pass a zero value for the JSMMETASIZE or JSMDATASIZE parameters.

The JSMX_BINDFLD API operates independently of the JSMX_BINDLST API – that is you may bind a set of non-recurring fields (using JSMX_BINDFLD), a list (using JSMX_BINDLST) or both.

Whether you need to use one, both or neither will depend on the function and requirements of the service and service command you are using.

Parameters

Number	Symbolic Name	Description
1	JSMHDL	Input, character 4. The JSMX_OPEN call returns the connection handle it assigns to this connection. This handle is used on subsequent JSMX_COMMAND and other API calls to identify the connection to which the call applies.
2	JSMMETA	Input, structure. Specifies a block of memory that contains zero or more field meta-data definitions corresponding to the structure of the fields provided in JSMDATA. You must provide one field meta-data definition for each field in the structure provided in

JSMDATA, and in the same order as the fields occur in JSMDATA.

Each meta-data entry must be formatted as follows:

Positions Description

1 – 50 Symbolic field name (50)

This name does not have to be the same as the name used in the RPG program. However, for some LANSAs Integrator services the symbolic name must match that expected by the service – this is particularly so for those services such as the XML and SOAP services that perform mapping between external data and program fields.

51 Data Type (1)

Supported types are A, P or S.

52 – 58 Field size (7)

This is the number of characters or digit positions, not the byte size.

59 – 60 Field decimal places. (2).

- | | | |
|---|--------------------|---|
| 3 | JSMMETASIZE | <p>Input, integer 10.</p> <p>Specifies the size of the memory block provided in the JSMMETA parameter in bytes. This must be a multiple of 60 (60 being the size of a single meta-data entry).</p> |
| 4 | JSMDATA | <p>Input/Output, structure.</p> <p>Specifies a block of memory that contains the fields described by the JSMMETA parameter.</p> <p>Depending upon the service and the service command, the field values might be used as input to the service, output from the service or both.</p> <p>When used as output from the service, the JSM will write the output field values to the memory identified by this parameter.</p> |
| 5 | JSMDATASIZE | <p>Input, integer 10.</p> <p>Specifies the size of the memory block provided in the JSMDATA parameter in bytes. This should normally be equal to the sum of the byte sizes of the fields described by the JSMMETA parameter.</p> |

- | | | |
|---|--------|--|
| 6 | JSMSTS | <p>Output, character (variable, recommended 20).</p> <p>The JSM server returns the status of the operation in this field. The client application can test this field to determine whether the operation succeeded. In most cases a status of 'OK' indicates successful completion.</p> |
| 7 | JSMMSG | <p>Output, character (variable, recommend 512).</p> <p>The JSM server may return a message concerning the operation in this field - for example a completion message or an error message when the operation fails.</p> |

3.4.4 JSMX_BINDLST

The JSMX_BINDLST API is optional and is used to "bind" a list to the specified connection handle so that the list entries can be sent or received by the subsequent JSMX_COMMAND API call.

Only one list may be bound at once – you must issue the JSMX_BINDLST call before the JSMX_COMMAND call(s) that uses it.

The JSMX_BINDLST call is not persistent – that is, it lasts only until the next JSMX_COMMAND call completes.

You must call JSMX_BINDLST again before a subsequent JSMX_COMMAND API call, if the service command requires a bound list. (The same is true for bound fields – see the JSMX_BINDFLD.)

To "unbind" a list from the specified connection handler, you can pass a null value (use *OMIT in ILE RPG) for the JSMMETA or JSMLIST parameters or pass a zero value for the JSMMETASIZE or JSMLISTSIZ parameters.

The JSMX_BINDLST API operates independently of the JSMX_BINDFLD API – that is you may bind a set of non-recurring fields (using JSMX_BINDFLD), a list (using JSMX_BINDLST) or both.

Whether you need to use one, both or neither will depend on the function and requirements of the service and service command you are using.

Parameters

No.	Symbolic Name	Description
1	JSMHDL	Input, character 4. The JSMX_OPEN call returns the connection handle it assigns to this connection. This handle is used on subsequent JSMX_COMMAND and other API calls to identify the connection to which the call applies.
2	JSMMETA	Input, structure. Specifies a block of memory that contains zero or more field meta-data definitions corresponding to the structure of the fields in each entry of the list provided in JSMLIST. Each meta-data entry must be formatted as

follows:

	Positions	Description
	1 – 50	Symbolic field name (50) This name does not have to be the same as the name used in the RPG program. However, for some LANSAs Integrator services the symbolic name must match that expected by the service – this is particularly so for those services such as the XML and SOAP services that perform mapping between external data and program fields.
	51	Data Type (1) Supported types are A, P or S.
	52 – 58	Field size (7) This is the number of characters or digit positions, not the byte size.
	59 – 60	Field decimal places (2).
3	JSMMETASIZE	Input, integer 10. Specifies the size of the memory block provided in the JSMMETA parameter in bytes. This must be a multiple of 60 (60 being the size of a single meta-data entry).
4	JSMLIST	Input/Output, structure. Specifies a block of memory that contains the list described by the JSMMETA parameter. Depending upon the service and the service command, the list entries might be used as input to the service, output from the service or both. When used as output from the service, the JSM will write the output entries to the memory identified by this parameter. In an ILE RPG program, the list is usually implemented as a multiple occurrence data structure.
5	JSMENTRYSIZE	Input, integer 10. Specifies the size of each list entry. This should normally be equal to the sum of the byte sizes of the fields described by the JSMMETA parameter.
6	JSMENTRYCOUNT	Input/Output, integer 10.

On entry, this specifies the number of entries provided by the client application in the list identified by the JSMLIST parameter. If the client application does not provide list entries for the service command, then the application should specify zero.

Upon return, if the service command normally writes entries to the list provided, then it will update this field with the number of entries written (up to the maximum specified by the JSMENTRYMAX parameter).

- | | | |
|---|-------------|---|
| 7 | JSMENTRYMAX | Input, integer 10.
Specifies the maximum number of list entries that can be held in the list specified by the JSMLIST parameter. |
| 8 | JSMSTS | Output, character (variable, recommended 20).
The JSM server returns the status of the operation in this field. The client application can test this field to determine whether the operation succeeded. In most cases a status of 'OK' indicates successful completion. |
| 9 | JSMMSG | Output, character (variable, recommend 512).
The JSM server may return a message concerning the operation in this field - for example a completion message or an error message when the operation fails. Some service commands may return data related to the service in this field. |

3.4.5 JSMX_COMMAND

The JSMX_COMMAND API is used to load and unload services and execute commands supported by the service.

The connection handle (JSMHDL) parameter identifies which connection is to execute the command – the connection handle must be first obtained by a call to JSMX_OPEN.

If the client application wishes to exchange variables or lists with the service, then it must call the JSMX_BINDFLD and/or JSMX_BINDLST APIs before calling JSMX_COMMAND.

Fields and lists bound using the JSMX_BINDFLD and JSMX_BINDLST APIs remain bound only for the duration of one JSMX_COMMAND call.

Parameters

Number	Symbolic Name	Description
1	JSMHDL	Input, character 4. The JSMX_OPEN call returns the connection handle it assigns to this connection. This handle is used on subsequent JSMX_COMMAND and other API calls to identify the connection to which the call applies.
2	JSMCMD	Input, character (variable, recommend 512). Specifies the service command that is to be executed by the loaded service or a SERVICE_LOAD or SERVICE_UNLOAD command to load or unload a service.
3	JSMSTS	Output, character (variable, recommended 20). The JSM server returns the status of the operation in this field. The client application can test this field to determine whether the operation succeeded. In most cases a status of 'OK' indicates successful completion.
4	JSMMSG	Output, character (variable, recommend 512). The JSM server may return a message concerning the operation in this field - for example a completion message or an error message when the operation fails.

Some service commands may return data related to the service in this field.

A service is loaded before commands to the service are executed. A service is unloaded once you are finished using it. Only one service can be loaded at a time. Refer to [3.7 Command](#).

The JSMX_COMMAND has a number of:

[RPGX Reserved Commands](#)

[RPGX Reserved Keywords](#)

For command services details, refer to [Java Service Manager Services](#).

Technical Note

For a service to receive HTTP posted content, the SERVICE_CONTENT(*HTTP) keyword is required on the SERVICE_LOAD command.

If this is the first connection to use the SERVICE_CONTENT(*HTTP) keyword, then the posted HTTP content is read and sent to the service.

This connection takes responsibility for sending the HTTP response.

If this is NOT the first connection to use the SERVICE_CONTENT(*HTTP) keyword, then this connection only takes responsibility for sending the HTTP response. The SERVICE_LOAD command does NOT receive the HTTP content, but does receive the HTTP keywords.

When the connection that is responsible for the HTTP response uses the JSMX_CLOSE to close the connection, the returned byte array response becomes the HTTP response.

Only one connection can have responsibility for sending the HTTP response at a time.

HTTP content can only be read once.

HTTP response can only be written once.

HTTP keywords are always sent with the SERVICE_LOAD command.

Scenario A

#1 JSMX_OPEN - open connection

#2 JSMX_OPEN - open connection

#3 JSMX_OPEN - open connection
#1 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)
SERVICE_CONTENT(*HTTP)") - send CGI keywords, read STDIN, claim
ownership
#2 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)") - send CGI
keywords
#3 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)
SERVICE_CONTENT(*HTTP)") - send CGI keywords, transfer ownership
#1 JSMX_CLOSE - close connection
#2 JSMX_CLOSE - close connection
#3 JSMX_CLOSE - close connection and write STDOUT

Scenario B

#1 JSMX_OPEN - open connection
#1 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)
SERVICE_CONTENT(*HTTP)") - send CGI keywords, read STDIN, claim
ownership
#1 JSMX_CLOSE - close connection and write STDOUT
#2 JSMX_OPEN - open connection
#2 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)") - send CGI
keywords
#2 JSMX_CLOSE - close connection
#3 JSMX_OPEN - open connection
#3 JSMX_COMMAND("SERVICE_LOAD SERVICE(xxx)
SERVICE_CONTENT(*HTTP)") - send CGI keywords, transfer ownership
#3 JSMX_CLOSE - close connection

RPGX Reserved Commands

Any command or keyword that starts with SERVICE_ is reserved.

For example, these commands are reserved:

SERVICE_LOAD	SERVICE(servicename) TRACE(option) TRACE_
SERVICE_SET	TRACE(*CLEAR)
SERVICE_GET	PROPERTY(property)
SERVICE_RECLAIM	
SERVICE_UNLOAD	

The SERVICE_LOAD command instructs the JSM service thread to load and instantiate the specified service program. Only one service class is loaded at a time. The SERVICE_LOAD command is a good place for the Java service programmer to write the code to load default values being used with the current service.

The SERVICE_SET command allows the enabling of trace file clearing.

The SERVICE_GET command allows access to the keyword/value properties from the associated service properties file. To reduce the conflict between user-defined keyword names and current and future LANSAs names prefix the keyword with a global unique name. The service property keyword is case insensitive.

The service property keyword value is returned in the #JSMMSG field.

If the keyword does not exist, then a status of NOT_EXIST is returned.

The special PROPERTY values *SERVICE and *SERVICECLASS return the loaded service class name.

Example

```
com.acme.property.messageType=html
#
message.001=Service successfully loaded
message.002=Service successfully unloaded
message.003=Command is not supported :
```

The SERVICE_RECLAIM command allows an explicit garbage collection to be done.

You could use this command in conjunction with the JVM `-Xcompactexplicitgc` option.

The `SERVICE_UNLOAD` command instructs the JSM service thread to invalidate the currently loaded service program and expect to load another service. This command is optional and does not need to be called if the JSM connection is going to be closed. (The `SERVICE_UNLOAD` command is a good place for the Java service programmer to put code to release currently allocated resources.)

RPGX Reserved Keywords

The following keywords are reserved:

SERVICE_CONTENT

TRIM

TRUNCATE

The SERVICE_CONTENT keyword is used on the SERVICE_LOAD command to trigger the exchange of content from the RPG program to the service being loaded. The only value supported by this keyword is *HTTP.

The service program performs a scan on the SERVICE_LOAD command string and searches for the following pattern:

SERVICE_CONTENT(*HTTP)

The keyword and its value must be identical to the example above, entirely in uppercase and without imbedded spaces.

3.4.6 JSMX_CLOSE

The JSMX_CLOSE API closes an open connection from the client application to the Java Service Manager.

The connection handle (JSMHDL) parameter identifies which connection is to be closed.

Parameters

Number	Symbolic Name	Description
1	JSMHDL	Input, character 4. The JSMX_OPEN call returns the connection handle it assigns to this connection. This handle is used on subsequent JSMX_COMMAND and other API calls to identify the connection to which the call applies.
2	JSMSTS	Output, character (variable, recommended 20). The JSM server returns the status of the operation in this field. The client application can test this field to determine whether the operation succeeded. In most cases a status of 'OK' indicates successful completion.
3	JSMMSG	Output, character (variable, recommended 512). The JSM server may return a message concerning the operation in this field - for example a completion message or an error message when the operation fails.

3.4.7 JSMX_END

The JSMX_END API is the last API call in a logical unit of processing.

This initializes/finalizes the internal state of the service program by closing any open connections and freeing any allocated resources. The internal state of the service program is scoped to the job.

You should not call this API while any JSM connections remain open if you wish to continue to use the open connection(s).

Parameters

No parameters.

3.4.8 Sample ILE RPGX Client Programs

The following RPG program example shows how use the HTTPService and the InboundFile handler to return a PDF document to a HTTP client program.

The program example also includes a binding fields and binding list example, which is not needed for its task of sending a HTTP response. Also the SERVICE_UNLOAD command is commented out so that the HTTP response return by the SEND command is returned to the HTTP client on the JSMX_CLOSE API call.

For further examples of ILE RPGX client applications, refer to [Client Application Examples](#).

Example

```
CRTRPGMOD MODULE(JSMBLD/DEMORPGSRV) SRCFILE(*LIBL/QC
```

```
CRTPGM PGM(JSMBLD/DEMORPGSRV) MODULE(*LIBL/DEMORPC  
BNDSRVPGM(*LIBL/JSMRPGSRV) ACTGRP(*CALLER)
```

```
D FieldDef1    S          60  DIM(6) CTDATA  
D FieldDef2    S          60  DIM(3) CTDATA  
D ListDef1     S          60  DIM(3) CTDATA  
D ListDefSize  S          10I 0 INZ(0)  
D ListEntSize  S          10I 0 INZ(0)  
D ListCount    S          10I 0 INZ(0)  
D ListMaxCount S          10I 0 INZ(0)  
D FieldDefSize S          10I 0 INZ(0)  
D FieldEntSize S          10I 0 INZ(0)  
*  
D JSMHDL       S           4  INZ(*BLANKS)  
D JSMSRV       S          100  INZ(*BLANKS)  
D JSMSTS       S           40  INZ(*BLANKS)  
D JSMMMSG      S         4096  INZ(*BLANKS)  
D JSMCMD       S         2048  INZ(*BLANKS)  
D ZEROLENGTH  S          10I 0 INZ(0)  
*  
D FLD1         DS  
D EMPLOYEE     50A  INZ('John Smith')
```

```

D SURNAME          30A INZ('Smith')
D SALARY           7P 2 INZ(30000.50)
D AGE              3S 0 INZ(25)
D DATE             D  DATFMT(*ISO)
D DATETIME         Z
*
D LST1            DS          OCCURS(10)
D EMPNME          50  INZ('Acme Corp')
D SURNME          35  INZ('Jones')
D GIVENME        20  INZ('Bill')
*
* Initialize
*
C          CALLB(D) 'JSMX_BEGIN'
C          PARM          *OMIT
C          PARM          ZEROLENGTH
*
* Open connection
*
C          CLEAR          JSMSRV
C          EVAL   JSMSRV = 'LOCALHOST:4560'
C          CALLB(D) 'JSMX_OPEN'
C          PARM          JSMHDL
C          PARM          JSMSRV
C          PARM          JSMSTS
C          PARM          JSMMSG
*
* Bind Fields
*
C          EVAL   DATE = %DATE()
C          EVAL   DATETIME = %TIMESTAMP()
*
C          EVAL   FieldDefSize = %SIZE(FieldDef1:*ALL)
C          EVAL   FieldEntSize = %SIZE(FLD1)
C          CALLB(D) 'JSMX_BINDFLD'
C          PARM          JSMHDL
C          PARM          FieldDef1
C          PARM          FieldDefSize
C          PARM          FLD1

```

```

C          PARM          FieldEntSize
C          PARM          JSMSTS
C          PARM          JSMMSG
*
* Bind List - set list count
*
C          EVAL    ListDefSize = %SIZE(ListDef1:*ALL)
C          EVAL    ListEntSize = %SIZE(LST1)
C          EVAL    ListCount  = 1
C          EVAL    ListMaxCount = %ELEM(LST1)
C          CALLB(D) 'JSMX_BINDLST'
C          PARM          JSMHDL
C          PARM          ListDef1
C          PARM          ListDefSize
C          PARM          LST1
C          PARM          ListEntSize
C          PARM          ListCount
C          PARM          ListMaxCount
C          PARM          JSMSTS
C          PARM          JSMMSG
*
* Send command
*
C          CLEAR          JSMCMD
C          EVAL    JSMCMD = 'SERVICE_LOAD' +
C                   ' SERVICE(HTTPSERVICE)' +
C                   ' SERVICE_CONTENT(*HTTP)' +
C                   ' TRACE(*YES)'
C          CALLB(D) 'JSMX_COMMAND'
C          PARM          JSMHDL
C          PARM          JSMCMD
C          PARM          JSMSTS
C          PARM          JSMMSG
*
* Bind List - use current value of list count
*
C          EVAL    ListDefSize = %SIZE(ListDef1:*ALL)
C          EVAL    ListEntSize = %SIZE(LST1)
C          EVAL    ListMaxCount = %ELEM(LST1)

```

```

C          CALLB(D) 'JSMX_BINDLST'
C          PARM          JSMHDL
C          PARM          ListDef1
C          PARM          ListDefSize
C          PARM          LST1
C          PARM          ListEntSize
C          PARM          ListCount
C          PARM          ListMaxCount
C          PARM          JSMSTS
C          PARM          JSMMSG
*
* Send command
*
C          CLEAR          JSMCMD
C          EVAL    JSMCMD = 'SEND' +
C                   ' HANDLER(InboundFile)' +
C                   ' FILE(test-input/ValidatingXML.pdf)' +
C                   ' CONTENT(*PDF)' +
C                   ' UACACHE(5)'
C
C          CALLB(D) 'JSMX_COMMAND'
C          PARM          JSMHDL
C          PARM          JSMCMD
C          PARM          JSMSTS
C          PARM          JSMMSG
*
* Send command
*
*          CLEAR          JSMCMD
*          EVAL    JSMCMD = 'SERVICE_UNLOAD'
*          CALLB(D) 'JSMX_COMMAND'
*          PARM          JSMHDL
*          PARM          JSMCMD
*          PARM          JSMSTS
*          PARM          JSMMSG
*
* Close service
*
C          CALLB(D) 'JSMX_CLOSE'

```

```
C          PARM          JSMHDL
C          PARM          JSMSTS
C          PARM          JSMMSG
*
* Finalize
*
C          CALLB(D) 'JSMX_END'
*
C          SETON          LR
```

**CTDATA FieldDef1

```
EMPLOYEE          A000005000
SURNAME           A000003000
SALARY            P000000702
AGE               S000000300
DATE              A000001000
DATETIME         A000002600
```

**CTDATA FieldDef2

```
EMPLOYEE          A000005000
SURNAME           A000003500
GIVENAME         A000002000
```

**CTDATA ListDef1

```
EMPLOYEE          A000005000
SURNAME           A000003500
GIVENAME         A000002000
```

3.5 C Client Error Messages

On Windows and Linux, client-side error responses are returned to 3GL programs in the form "JSMMSG nnn", where nnn is one of the numbers listed below.

JSMMSG	Description
---------------	--------------------

847	Cannot get job attributes.
848	Cannot create socket.
849	Cannot find server.
850	Cannot connect to server.
851	Service closed.
852	Service not opened.
853	Cannot send command, connection closed.
854	Cannot send command, connection closed.
855	Cannot send byte array, connection closed.
856	Cannot read protocol properties.
857	Cannot send keyword list, connection closed.
858	Bytearray size exceeds stdout storage limit.
859	Cannot send working list, connection closed.
860	Cannot send client information, connection closed.
861	Cannot send field list, connection closed.
862	Cannot send field data, connection closed.
863	Cannot allocate storage for field array.
864	Cannot allocate storage for field data.
865	Received status value is too long.
866	Received message value is too long.

- 867 Cannot read status value.
- 868 Cannot read message value.
- 869 Cannot allocate storage for byte array.
- 870 Cannot read byte array.
- 871 Cannot allocate storage for working list.
- 872 Cannot read working list.
- 873 Cannot allocate storage for field list.
- 874 Cannot read field list.
- 875 Cannot allocate storage for running checks.
- 876 Inbound field list check failure.
- 877 Cannot update working list.
- 878 Cannot update field data.
- 879 Cannot create conversion descriptions.
- 880 Cannot read stdin.

3.6 Tracing

JSM tracing is very comprehensive. When you first start to develop your applications using JSM, it will be very helpful if you know how to use the tracing facilities.

All tracing output is stored in the trace sub-directory below the JSM instance directory.

Standard output and Standard error:

IBM i If the shipped STRJSM program is used, then the standard output and standard error will be printed to the JSM output queue located in the JSM library.

Windows If the shipped STRJSM program is used, then the standard output and standard error will be saved to the JSM instance as
Linux STDOUT.TXT and STDERR.TXT.

A sub-directory named after the platform's job/process number is created to receive the trace file for the current JSM instance when tracing is requested.

The CLRJSM TRACEDIR(*YES) command allows the trace files and subdirectories to be removed. This command should be run when the instance is not running.

[3.6.1 Manager Tracing](#)

[3.6.2 Client Command Tracing](#)

[3.6.3 Service Program Tracing](#)

[3.6.4 Service Program Tracing from the Client](#)

[3.6.5 How Do I Start Tracing?](#)

[3.6.6 Where Do I Find Trace Files?](#)

[3.6.7 How Do I Clear Trace Files?](#)

3.6.1 Manager Tracing

Manager tracing allows the JSM to output information about the instance that is being started. This information reports the instance configuration parameters in use, but its main use is to report the properties of the [JVM](#) being used.

When manager tracing is enabled, STDOUT and STDERR are directed to the trace files STDOUT.TXT and STDERR.TXT instead of a spool file in JSM output queue. Java Virtual Machine and instance information is logged to a MANAGER.TXT file.

To turn on Manager tracing, uncomment the trace.manager keyword in the manager.properties file.

```
trace.manager=*yes
```

To turn off Manager tracing, you can either comment the trace.manager keyword or change the value from *yes to *no.

Reminder: In order for changes to the manager.properties file to take effect, you must stop and start the JSM. The manager properties are loaded when the JSM starts.

Example trace directory listing

```
Directory: /jvm/instance/trace/123456
```

```
MANAGER.TXT      STMF
STDOUT.TXT       STMF
STDERR.TXT       STMF
2005-06-22      DIR
```

```
Directory: /jvm/instance/trace/123456/2005-06-22
```

```
CLIENT00000001  DIR
CLIENT00000002  DIR
CLIENT00000003_NAME  DIR
```

3.6.2 Client Command Tracing

To trace the commands being sent from the client program to the JSM instance, you need to uncomment the `trace.transport.address`, `trace.transport.error.address`, `trace.service.address` or `trace.service.error.address` keyword in the `manager.properties` file.

The property `trace.transport.address` will turn on transport tracing for all services.

The property `trace.transport.error.address` will turn on transport tracing for all services and if no errors occur then the trace files will be deleted.

The property `trace.service.address` will turn on transport and service tracing for all services.

The property `trace.service.error.address` will turn on transport and service tracing for all services and if no errors occur then the trace files will be deleted.

The format of this keyword is:

```
trace.transport.address=<value>,<value>
trace.transport.error.address=<value>,<value>
trace.service.address=<value>,<value>
trace.service.error.address=<value>,<value>
```

where `<value>` is the client IP address that needs to be traced, as in this example:

```
trace.transport.address=137.76.2.202
trace.transport.error.address=137.76.2.202
trace.service.address=137.76.2.202
trace.service.error.address=137.76.2.202
```

or `*ALL`, to trace all clients, as in this example:

```
trace.transport.address=*all
trace.transport.error.address=*all
trace.service.address=*all
trace.service.error.address=*all
```

When a client program connects to the JSM instance and transport or service tracing is enabled, trace files are created in the JSM instance *trace/jobnumber/date/CLIENTnnnnnnnn* sub-directory.

The client directory sequence number is a unique incremental number assigned to each client thread.

The client communication trace records are written to a TRANSPORT.TXT file and the service trace records are written to a SERVICE.TXT file in this client trace directory.

Reminder: In order for changes to the manager.properties file to take effect, you must stop and start the JSM. The manager properties are loaded when the JSM starts.

Example trace directory listing

Directory: /jvm/instance/trace/123456

MANAGER.TXT	STMF
STDOUT.TXT	STMF
STDERR.TXT	STMF
2005-06-22	DIR

Directory: /jvm/instance/trace/123456/2005-06-22

CLIENT00000001	DIR
CLIENT00000002	DIR
CLIENT00000003_NAME	DIR

Example file listing of a client trace sub-directory

Directory: /jvm/instance/trace/123456/2005-06-22/CLIENT00000001

SERVICE.TXT	STMF
TRANSPORT.TXT	STMF

3.6.3 Service Program Tracing

Client command tracing does not need to be enabled for service program tracing to work.

To allow tracing into the service program being called by a client program, add an entry to the service.properties file.

The structure of this entry is:

```
trace.<service>=<value>
```

where <service> is the service name and <value> is the client IP address that needs to be traced, as in this example:

```
trace.FTPService=137.76.2.202
```

or *ALL, to trace all clients using the service, as in this example:

```
trace.FTPService=*all
```

When service tracing is enabled, a JSMTrace object is available from the JSMContainer object.

Normally this JSMTrace object is null. If the trace object is not null then text messages can be written to the associated SERVICE.TXT trace file.

Reminder: In order for changes to the service.properties file to take effect, you must stop and start the JSM or use the [Java Service Manager Console](#) or the [Integrator Studio](#) to refresh the loaded services.

3.6.4 Service Program Tracing from the Client

To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.

The possible values for the TRACE keyword are:

- *NO
- *YES
- *ERROR

The trace option *ERROR will turn on tracing and if the service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.

The optional TRACE_NAME keyword allows the client to append a user-defined name to the end of the client trace sub-directory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

During the service loading, tracing can be enabled by using this command:

```
SERVICE_LOAD SERVICE(servicename) TRACE(option) TRACE_NAME(i
```

3.6.5 How Do I Start Tracing?

Tracing is controlled by the entries in the `manager.properties` and `services.properties` files which are used by the Java Service Manager. You must edit these files to turn tracing on and off. Remember, the JSM must be restarted before the changes to these files will take effect.

Refer to the type of tracing you wish to carry out:

[3.6.1 Manager Tracing](#)

[3.6.2 Client Command Tracing](#)

[3.6.3 Service Program Tracing](#)

[3.6.4 Service Program Tracing from the Client.](#)

3.6.6 Where Do I Find Trace Files?

All tracing output is stored in the trace sub-directory below the JSM instance directory.

To view the trace file, use the [Work with Java Service Manager \(WRKJSM\)](#) option on the JSM Menu described in [Java Service Manager Administration for IBM i](#).

A new trace directory is created each time the JSM is started, provided the relevant service property is enabled.

For example, if the JSM job number is 123456, then the directory would appear as follows:

```
/jsm/instance/trace/123456
```

Example trace directory listing

```
Directory: /jsm/instance/trace/123456
```

```
MANAGER.TXT      STMF
STDOUT.TXT       STMF
STDERR.TXT       STMF
2005-06-22       DIR
```

```
Directory: /jsm/instance/trace/123456/2005-06-22
```

```
CLIENT00000001  DIR
CLIENT00000002  DIR
CLIENT00000003_NAME  DIR
```

Example file listing of a client trace sub-directory

```
Directory: /jsm/instance/trace/123456/2005-06-22/CLIENT00000001
```

```
SERVICE.TXT      STMF
TRANSPORT.TXT    STMF
```

If you have changed the tracing properties in the manager properties files, then you must stop and restart the JSM in order for the changes to take effect. When you restart the JSM, it will start with a new job number and a new directory will appear in your trace sub-directory.

If manager tracing is turned on, `MANAGER.TXT`, `STDOUT.TXT` and `STDERR.TXT` files are created in the trace job number sub-directory when the JSM instance starts.

If client tracing is enabled, a `CLIENTnnnnnnnn` directory is created where `nnnnnnnn` is a sequential number. This directory contains the transport and command information that is sent by the JSM client program. An optional trace name can be appended to the end of this directory name using the `TRACE_NAME` keyword on the `SERVICE_LOAD` command.

It is recommended that you clear trace files regularly. Refer to [Clear Java Service Manager \(CLRJSM\)](#).

3.6.7 How Do I Clear Trace Files?

Tracing can generate a large number of files which may need to be cleared once you have finished your review of the tracing files.

- On IBM i you may use the CLRJSM TRACEDIR(*YES) command to delete the trace files and subdirectories.
Refer to [Clear Java Service Manager \(CLRJSM\)](#).
- On Windows select Clear Trace from the Start Programs menu.
Do not run this command when JSM tracing is active as current trace data will be lost.
Refer to Clear Trace in [Java Service Manager Administration for Windows](#).
- On Linux the trace files are cleared using the "-c" option when starting the JSM instance: "strjasm -c".
Refer to Clear Trace Directory in [Java Service Manager Administration for Linux](#).
- Integrator Studio can also clear the trace files.

It is recommended that you clear your trace files regularly.

3.7 Command

The primary means of communicating with a JSM service is by sending a command string.

The command string consists of a command name and zero or more keywords.

COMMAND KEYWORD1(VALUE) KEYWORD2(VALUE)

The command string is parsed by the JSM service.

- Command name is converted to uppercase
- Keywords are converted to uppercase
- Keyword values keep the same case
- Keyword values have leading and trailing blanks removed
- Keywords with a blank value are ignored

Special handling of keyword values is possible by using double quotes.

If a value requires leading and or trailing blanks to be preserved then double quote the value.

Any value that starts and ends with double quotes will have the double quotes removed.

Double quote the keyword value to protect open and close brackets within the keyword value.

If the value also contains double quotes and the entire string is double quoted then escape the double quote with a backslash.

Keyword Value	Parsed Value	
KEYWORD ()	null	keyword is ignored
KEYWORD ("")		empty
KEYWORD (" ")		spaces
KEYWORD (abc)	abc	
KEYWORD ("abc")	abc	

KEYWORD (" abc")	abc	
KEYWORD (ab"c)	ab"c	
KEYWORD ("ab\"c")	ab"c	
KEYWORD ("a(b)c")	a(b)c	
KEYWORD ("a(\"b\")c")	a("b")c	

When a command string is printed out in the SERVICE.TXT or TRANSPORT.TXT trace files, the keyword password values are hidden by five stars (*****).

Any command keyword named PASSWORD or ending with the string PASSWORD will have the keyword value hidden.

Any command keyword named MASTER will have the keyword value hidden.

The following command keywords will have their password component hidden.

- SIGNER (name:password)
- KEYSTORE (name:password)
- RECEIVER (name:password)
- RECIPIENT (name:password)

To view passwords in the trace files, add the trace.passwords=*yes property to the manager.properties file.

To view field information sent with the command, add the trace.fields=*yes property to the manager.properties file.

Each JSM client has different command capabilities.

RDML command [3.1.2 JSM_COMMAND](#)

RDMLX command [3.2.3 JSMX_COMMAND](#)

ILE RPGX command [3.4.5 JSMX_COMMAND](#)

If a JSMDirect function needs to access multiple services within the same program, then it is recommended to use multiple JSM connections.

It is possible for a single JSM connection to use multiple services sequentially by unloading and loading another service.

JSM
OPEN

SERVICE_LOAD SERVICE(servicename1)	Load specified service program
---------------------------------------	-----------------------------------

User-defined commands

SERVICE_UNLOAD	Unload service
----------------	----------------

SERVICE_LOAD SERVICE(servicename2)	Load specified service program
---------------------------------------	-----------------------------------

User-defined commands

SERVICE_UNLOAD	Unload service (optional)
----------------	---------------------------

JSM
CLOSE

The JSMStorage object is persistent between JSM OPEN and JSM CLOSE, so *servicename1* can put an object into the JSMStorage object and *servicename2* can get this saved object. Alternatively, you may write information to a file that can be shared between services.

3.8 Data Areas

On IBM i the Java Service Manager uses two data areas, JSMCLTDTA and JSMMGRDTA.

On Windows and Linux, these data areas are implemented as flat files, jsmcltdta.txt and jsmmgrdta.txt. Their structure is based on lines with keyword-value pairs. Lines starting with a # are comments and will not be processed.

1. JSMCLTDTA

If the JSMCLTDTA data area is not in the library list, then the default server and exclude prefix are set to LOCALHOST:4560 and JSM.

The data area layout is:

1-50	Default host name and port. For example: LOCALHOST:4560
51-60	LANSA field name exclude prefix. For example: JSM

If the jsmcltdta.txt file is not found, then the default server and exclude prefix are set to LOCALHOST:4560 and JSM.

The jsmcltdta.txt file is located and installed in <sysdir>.

The file keywords are:

DefaultServer	Default host name and port. For example: DefaultServer=LOCALHOST:4560
ExcludePrefix	LANSA field name exclude prefix. For example: ExcludePrefix=JSM

2. JSMMGRDTA

The JSMMGRDTA data area is updated with the JSM instance directory during the install process.

The data area layout is:

1-50	Default instance path. For example: /LANSA_xxxPGMLIB/jsm/instance
------	--

The jsmmgrdta.txt file is shipped with reasonable default values.

The file keywords are:

Priority	Process priority. For example: Priority=NORMAL
Instance	Instance path. For example: Instance=..\JSMInstance
JVM	Path to Java program. For example: JVM=java
JVMOptions	Java program options. For example: -Xms128M -Xmx128M -Xrs - Djava.endorsed.dirs=.\endorsed

3.9 Resource properties - Studio sections

The Studio client application allows sections of resource property files to be edited.

When Studio creates a new project a unique project id is generated using the current date time.

When the Studio application publishes the project entries to the instance server, the resource properties file replaces the current studio-project section (Identified by the id attribute) and inserts the new data. If a studio-project section is not found a new section is appended to the end of the file.

A studio-project section with id="20000000-000000" is reserved by LANSA.

A studio-project section can be created manually using a text editor. The following example illustrates a manual entry with an id="20030101-000000" to hold some override property entries.

All property file entries must exist within a studio-project section, during an upgrade process any entries found outside a section will be placed in the studio-project id="00000000-000000" name="unassigned" section. They should be assigned to another section or deleted.

The following example illustrates how the shipped manager.properties file has had an extra section added to the bottom. The final tcp.port=4562 will be used, also manager and client tracing have been enabled. When LANSA Integrator is upgraded the upgrade process will replace the studio-project id="20000000-000000" with a new section but other sections in the properties file will remain. The section replacement will occur at the current location of the old section, so section order is maintained.

It is best practice not to modify entries in the LANSA studio section but add your own section to the bottom of the file. These entries will remain untouched during an upgrade.

```
#!<studio-project id="20000000-000000" name="lansa">
#
# Java Service Manager configuration
#
tcp.port=4560
tcp.backlog=20
tcp.interface=*all
#
```

```
# console.tcp.port=4561
# console.tcp.backlog=5
# console.tcp.interface=*all
#
# trace.manager=*yes
# trace.transport.address=*all
# trace.transport.error.address=*all
# trace.service.address=*all
# trace.service.error.address=*all
#
#!</studio-project>
#
#!<studio-project id="20030101-000000" name="user-override">
#
tcp.port=4562
tcp.backlog=20
tcp.interface=*all
#
console.tcp.port=4563
console.tcp.backlog=5
console.tcp.interface=*all
#
trace.manager=*yes
trace.service.address=*all
#
#!</studio-project>
```

3.10 Resource properties - Internationalization

If you are executing the JSM in a distributed or tiered architecture, the remote servers may exist in different countries. These servers may have specific regional or language requirements.

The Java Service Manager resource files are internationalized.

The resource files have a load hierarchy.

```
basefile.properties
basefile_language.properties
basefile_language_country.properties
```

Each property entry read will replace a previous property entry. This replacement can occur due to a duplicate key entry further down in the same file or duplicate entry in the later locale property files.

The [locale](#) is determined from the client program. A program running on a IBM i machine will supply its encoding and locale information to the JSM server.

Example

```
CCSID      037
LANGUAGEID ENG
COUNTRYID  AU
```

In this case, the JSM server will create a thread to handle the connection. This thread will use encoding Cp037 and locale EN_AU.

A single JSM server can handle multiple clients with different encoding and locale information.

4. Java Service Manager HTTP Extensions

If the LANSAs function/RPG program is being executed from a request over the Internet (i.e. it is invoked by a CGI program from an HTTP server), then the JSM Client is a LANSAs function/RPG program started via JSMDirect.

[4.1 JSMDirect](#)

[4.2 JSMProxy](#)

[4.3 JSMAdmin](#)

[4.4 Location of the LANSAs System \(IBM i\)](#)

[4.5 Location of the LANSAs System \(Windows\)](#)

[4.6 Location of the LANSAs System \(Linux\)](#)

[4.7 Exit Programs \(IBM i\)](#)

[4.8 Deployment \(IBM i\)](#)

[4.9 Log Files \(Windows\)](#)

[4.10 Apache Directives](#)

[4.11 Apache URL Rewriting](#)

[4.12 Apache Reverse Proxy](#)

[4.13 Apache SSL Support](#)

[4.14 Apache Tracing \(IBM i\)](#)

[4.15 SSL Support](#)

[4.16 SSL Handshake](#)

[4.17 JSM HTTP Server](#)

For details about installing the JSM Manager, JSMDirect and JSMProxy, refer to [Install Other Features](#) in the Installing LANSAs on Windows Guide.

4.1 JSMDirect

JSMDirect is an HTTP service extension to JSM.

JSMDirect allows a LANSAs/RPG program to participate in Web services and [JVM](#) transactions.

The JSMDirect program accepts POSTed content that contains the Content-Length HTTP protocol property. The IBMi JSMDirect program running on the IBMi Apache server can also accept Chunked-Transfer encoded content.

Note: Chunked-Transfer encoding is not supported on the Windows platform.

The JSMDirect query string can be in the original keyword format or the industry standard name-value pairs. The name service is reserved.

To display the JSMDirect version number invoke the program with no arguments. An HTML About page will be returned.

`http://mycompany/cgi-bin/jsmdirect`

`http://mycompany/cgi-bin/jsmdirect.exe`

Using JSMDirect, a LANSAs function can be invoked as follows:

`http://mycompany/cgi-bin/jsmdirect?appname`

`http://mycompany/cgi-bin/jsmdirect?service=appname`

`http://mycompany/cgi-bin/jsmdirect.exe?appname`

`http://mycompany/cgi-bin/jsmdirect.exe?service=appname`

For example, to run the ORDERENTRY function:

`http://mycompany/cgi-bin/jsmdirect?orderentry`

`http://mycompany/cgi-bin/jsmdirect?service=orderentry`

`http://mycompany/cgi-bin/jsmdirect.exe?orderentry`

`http://mycompany/cgi-bin/jsmdirect.exe?service=orderentry`

The JSMDIRECT program locates the LANSAs function to be executed based on the *appname*.

JSMDirect requires an installed HTTP Server.

JSMDirect does not require or use any of the LANSAs for the Web software.

4.1.1 WSDL Option

JSMDirect has the ability to return the contents of the file specified in the WSDL file location column of DC@W29 or dc_w29.txt configuration file.

By including an additional '+wsdl' parameter on the web service URL, JSMDirect will return the contents of the file instead of executing the web service.

Example

```
http://mycompany.com/cgi-bin/jsmdirect?mywebservice+wsdl
```

The file content is treated as binary and text must be UTF-8 encoded.

The IBM i Apache HTTP Server must be configured to run in BINARY mode.

By returning HTML content you can offer more selection choices to the end user, instead of just WSDL content.

JSMDirect recognizes the following file extensions and uses an appropriate content type. If the file extension is not supported then the content type of unknown/unknown is used.

htm	text/html; charset=utf-8
html	text/html; charset=utf-8
xhtml	application/xhtml+xml; charset=utf-8
wsdl	application/xml; charset=utf-8
	unknown/unknown

WSDL and unknown content will also include a Content-Disposition header using the name of the file, this allows the HTTP browser to prompt the user with a save file dialog.

Example

Content-Disposition: attachment; filename=xxxx

Example: HTML

```
<html>
```

```
<head>
```

```
<title>MyWebService</title>
```

```
</head>
```

```
<body>
```

This is MyWebService in HTML format

Use the following link to download the PDF documentation

Use the following link to download the WSDL file


```
</body>
```

```
</html>
```

Example: XHTML

```
<!DOCTYPE html PUBLIC "-
```

```
//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/L  
transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
```

```
<title>MyWebService</title>
```

```
</head>
```

```
<body>
```

This is MyWebService in XHTML format

Use the following link to download the PDF documentation

Use the following link to download the WSDL file

</body>

</html>

4.1.2 SERVICE_LOAD

The JSM client is aware that it has been invoked by the JSMDirect program and will modify the SERVICE_LOAD command being sent to the JSM service manager.

Normally, the SERVICE_LOAD command would be:

```
SERVICE_LOAD SERVICE(servicename)
```

When invoked by JSMDirect the modified SERVICE_LOAD command is:

```
SERVICE_LOAD SERVICE(servicename)
                REQUEST-METHOD(method)
                CONTENT-TYPE(mime type)
                REMOTE-ADDRESS(address)
                REMOTE-USER(user)
                SERVER-NAME(name)
                SERVER-PORT(port)
                CGI-INPUT-MODE(mode)
                CGI-OUTPUT-MODE(mode)
                CGI-ASCII-CCSID(ccsid)
                CGI-EBCDIC-CCSID(ccsid)
                SERVER-SOFTWARE(software)
                SERVER-PROTOCOL(version)
                GATEWAY-INTERFACE(version)
                USER-AGENT(useragent)
                USER-AGENT-PATH(path)
                USER-AGENT-FILE(file)
                USER-AGENT-LOCALE(locale)
```

QUERY-STRING(querystring)
SOAPACTION(soapaction)
COOKIE(cookie data)
ACCEPT(mime types)
REFERER(url)
DIRECT-VERSION(version)
PROXY-VERSION(version)
PROXY-SERVER-NAME(name)
PROXY-SERVER-PORT(port)
PROXY-REMOTE-ADDRESS(address)
PROXY-REMOTE-USER(user)
AS2-TO(company)
AS2-FROM(company)
AS2-VERSION(version)
FROM(from)
DATE(date)
SUBJECT(subject)
MESSAGE-ID(message-id)
DISPOSITION-NOTIFICATION-TO(address)
DISPOSITION-NOTIFICATION-OPTIONS(options)
RECEIPT-DELIVERY-OPTION(url)

If information is being sent to the function (i.e. if the HTTP request is a POST), then the information (content payload) is passed with the command as the byte array object.

The service program can access the HTTP POST content by calling the `getByteArray` method on the command object.

When the JSM CLOSE is performed, the last byte array returned by a JSMResponse object is sent back to the HTTP user agent.

Example of some HTTP keyword values:

PROPERTY	EXAMPLE VALUE
USER-AGENT-PATH	/upload/data
CONTENT-TYPE	text/comma-separated-values; charset=utf-8
CGI-EBCDIC-CCSID	37
CGI-ASCII-CCSID	819
CGI-INPUT-MODE	BINARY
CGI OUTPUT MODE	BINARY
SERVER-PORT	80
USER-AGENT-FILE	order.xls
REQUEST-METHOD	POST
REMOTE-USER	CUST100
USER-AGENT	LANSA Integrator UserAgent/11.4.0 (Client)
REMOTE-ADDRESS	10.2.1.7
SERVER-PROTOCOL	HTTP/1.0
QUERY-STRING	Order
SERVER-NAME	lansa01.lansa.com.au
SERVER-SOFTWARE	Apache
USER-AGENT-LOCALE	en_AU

4.1.3 JSMDirect Set up on IBM i

The Apache server can be configured to run CGI programs in %%BINARY/MIXED%%, %%BINARY/BINARY%% or BINARY mode.

The preferred mode is BINARY.

By default, the JSMDirect CGI program runs under the user profile QTMHHTTP1.

Following is an example IBM Apache Server Configuration:

```
# Apache Configuration - JSM Services
Options None
Listen 10.2.0.170:1099
ServerRoot /www/jsmapache
DocumentRoot /www/jsmapache/htdocs
# DefaultFsCCSID 37
# DefaultNetCCSID 819
# ServerUserID USERPROFILE
#
LogLevel Warn
LogCycle Daily
ErrorLog logs/error_log
CustomLog logs/access_log combined
LogFormat "%{User-agent}i" agent
LogFormat "%{Referer}i -> %U" referer
LogFormat "%{Cookie}n \"%r\" %t" cookie
LogFormat "%h %l %u %t \"%r\" %>s %b" common
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-
Agent}i\"" combined
#
SetEnvIf "User-Agent" "Mozilla/2" nokeepalive
SetEnvIf "User-Agent" "JDK/1\0" force-response-1.0
SetEnvIf "User-Agent" "Java/1\0" force-response-1.0
SetEnvIf "User-Agent" "RealPlayer 4\0" force-response-1.0
SetEnvIf "User-Agent" "MSIE 4\0b2;" nokeepalive
SetEnvIf "User-Agent" "MSIE 4\0b2;" force-response-1.0
#
# StartCGI 1
```

```
# SendBufferSize 32768
# ReceiveBufferSize 32768
#
# Use name-based virtual hosting
NameVirtualHost 10.2.0.170
#
<VirtualHost 10.2.0.170>
# The first virtual host directive will become the default host
# This traps the use of the IP address, unsupported or no host name
# It also has no authority to access the document root directory
Options None
ServerName 10.2.0.170
DocumentRoot /www/jsmapache/htdocs
</VirtualHost>
#
<VirtualHost 10.2.0.170>
Options None
ServerName LANSA01
ServerAlias LANSA01.LANSA.COM.AU
DocumentRoot /www/jsmapache/htdocs-site1
CGIConvMode BINARY
ScriptAliasMatch ^/cgi-
bin/jsmpoxy(.*) /qsys.lib/devjms.lib/jsmpoxy.pgm$1
ScriptAliasMatch ^/cgi-
bin/jsmdirect(.*) /qsys.lib/devjms.lib/jsmdirect.pgm$1
Timeout 3000
#
<Directory /www/jsmapache/htdocs-site1>
Options None
Order Allow,Deny
Allow from all
AllowOverride None
</Directory>
#
<Directory /qsys.lib/devjms.lib>
Options None
Order Allow,Deny
Allow from all
AllowOverride None
```

```
# Require valid-user
# AuthType Basic
# AuthName "Restricted Service"
# UserID QTMHHTTP1
# PasswdFile %%SYSTEM%%
</Directory>
</VirtualHost>
#
<VirtualHost 10.2.0.170>
Options None
ServerName STUDIOADMIN
DocumentRoot /www/jsmapache/htdocs-site2
CGIConvMode BINARY
ScriptAliasMatch ^/cgi-
bin/jsmadmin(.*)/qsys.lib/devjsm.lib/jsmadmin.pgm$1
#
<Directory /www/jsmapache/htdocs-site2>
Options None
Order Allow,Deny
Allow from all
AllowOverride None
</Directory>
#
<Directory /qsys.lib/devjsm.lib>
Options None
Order Allow,Deny
Allow from 10.2.1.46
Allow from 10.2.1.47
Allow from 10.2.1.48
AllowOverride None
</Directory>
</VirtualHost>
#
<VirtualHost 10.2.0.170>
Options +Includes
ServerName LANSA01WEB
DocumentRoot /LANSA_xyzpgmlib/webserver/www/htdocs
ServerUserID XYZPGMLIB
CGIConvMode EBCDIC
```

```
Alias /images /LANSA_xyzpgmlib/webserver/images
ScriptAliasMatch ^/cgi-
bin/lansaweb(.*) /qsys.lib/xyzcomlib.lib/lansaweb.pgm$1
TimeOut 3000
#
<Directory /qsys.lib/xyzcomlib.lib>
  Order Allow,Deny
  Allow from all
</Directory>
#
<Directory /LANSA_xyzpgmlib/webserver/images>
  Order Allow,Deny
  Allow from all
</Directory>
#
<Directory /LANSA_xyzpgmlib/webserver/www/htdocs>
  Order Allow,Deny
  Allow from all
</Directory>
</VirtualHost>
#
# Global server directives
#
<Directory />
  Options None
  Order Allow,Deny
  Deny from all
  AllowOverride None
</Directory>
#
<Directory /www/jsmapache/htdocs>
  Options None
  Order Allow,Deny
  Deny from all
  AllowOverride None
</Directory>
```

Registering Functions for JSMDirect

On IBM i the JSMDirect program locates the LANSAs function to be executed by performing a keyed lookup on file DC@W29V1.

The Host Name is the host component of the request URL this is determined by the HTTP protocol Host keyword.

If the URL is `http://www.lansa.com/cgi-bin/jsmdirect` then the Host Name is `WWW.LANSA.COM:80`.

If the URL is `http://10.1.2.3/cgi-bin/jsmdirect` then the Host Name is `10.1.2.3:80`.

This allows multi-homing where the IP address of `www.lansa.com` and `10.1.2.3` are the same but the HTTP client program specifies the host name in the HTTP protocol allowing the HTTP server to treat `www.lansa.com` and `10.1.2.3` as different hosts.

A Host Name of `*DEFAULT` means any host.

JSMDirect first does a keyed lookup for on the Application Name and Host Name, if an entry does not exist then a second keyed lookup on Application Name and `*DEFAULT` is done.

To update the DC@W29 file use LANSAs Integrator Studio or you may use Data File Utility or SQL commands on the file.

Refer to [Maintaining Registration File DC@W29](#).

Structure of file DC@W29:

K W29SRV	Application Name	Name identifying the application to be executed. This is the name entered as part of the URL.
K W29LHO	Host Name	Host Name specified either as <code>*DEFAULT</code> or <code>HostName:Port</code> .
W29MOD	Process	LANSAs process.
W29FMT	Function	LANSAs function to be executed.
W29P#I	Partition	LANSAs partition. The process and function must be located in the specified partition.
W29PLN	Language	Execution language for the function.
W29PGM	Program	3GL program to be executed. If a 3GL program is

specified, then the process, function, partition and language fields should be blank.

W29RDX	RDMLX	Flag function as an RDML or RDMLX function.
W29WSD	WSDL Path	Location of WSDL file.

Note: The language code is an optional value in the registration of functions in the IBM i DC@W29 but is a required one in dc_w29.txt on Windows. If you don't register the language code in DC@W29, LANSA will run the server function using the partition's default language. If you don't register the language code in dc_w29.txt, JSMDirect will end with an error.

Example (without Program parameter):

```
ORDERENTRY MYCOMPUTER:80 JSERVICE JS015 WIZ ENG
ORDERENTRY *DEFAULT JSERVICE JS016 WIZ ENG
```

Maintaining Registration File DC@W29

To display and/or update the DC@W29 registration file use LANSAs Integrator Studio or maintenance program DC@P8700. You may also use Data File Utility or SQL commands to update this file.

The DC@P8700 maintenance program can be called either in interactive or batch mode, as follows:

To call in interactive mode:

From a command line

CALL DC@P8700

To call in batch mode or from another program:

Parameters must be supplied to identify the action to be performed and service.

Note: If no parameters are supplied, an Action of DSP is assumed.

Following is a list of possible parameters:

Parameter	Type & length	Values	Description
Action	A(3)	DSP ADD UPD DLT	Display a service. Add a service. Update a service. Delete a service.
Service	A(30)		Service name.
Host	A(80)		Host name.
Return Code	A(2)	OK ER	Action performed successfully. Error in action requested.
Type of Target	A(3)	FUN 3GL	Target is a Process/Function. Target is a Program.
Target	A(32)		This may contain either: Process A(10) Function A(7) Partition A(3) Language A(4) or Program A(32)

RDMLX Flag	A(1)	Y N blank	If Target is a Process/Function Y=Function is RDMLX N=Function is not RDMLX (A blank is an implied N).
WSDL Path	A(256)		Absolute path of WSDL file.

The parameters that are required depend on the Action code:

For this Action	These parameters are required
DSP	Action
ADD	Action Service Host Return Code Type of Target Target RDMLX Flag WSDL Path
UPD	Action Service Host Return Code Type of Target Target RDMLX Flag WSDL Path
DLT	Action Service Host Return Code

JSMDirect Program Logic

- Remote client program using HTTP protocol connects to HTTP server.
- HTTP server executes JSMDIRECT program. (Look for messages in HTTP server job log).
- JSMDirect program checks for arguments, if no arguments sends an about HTML page to client program.
- JSMDirect program calls exit program JSMDRTEXT (if present) with an EVENT parameter value of ENTRY.
- JSMDirect program uses the service argument value and the SERVER_HOST and SERVER_PORT environment variable values to do a keyed lookup on DC@W29V1 to determine the LANSA function or 3GL program associated with the service. If no record is found then another keyed lookup is done using the service argument value and the host value of *DEFAULT. If no record is found then an error message is returned and logged via the JSMDRTEXT program with the EVENT parameter set to an ERRnnnn value. (Look for messages in HTTP server job log).
- JSMDirect program reads standard input.
- JSMDirect program calls exit program JSMLSAEXT (if present) or internally calls LANSA.
- JSMDirect program waits for the LANSA function or the 3GL program to complete.
- JSMDirect program writes function or program response to standard output.
- JSMDirect program calls exit program JSMDRTEXT (if present) with an EVENT parameter value of EXIT.

4.1.4 JSMDirect Set up on Windows

If IIS is installed on the target PC, the install/upgrade will automatically configure IIS so that the physical folder JSMCGI is mapped to the virtual folder cgi-bin.

If IIS is not installed, you will need to manually configure the Web Server.

You must manually specify the Windows logon for executing the jsmdirect.exe CGI program.

By default, IIS would use the default Windows logon IUSR_<machinename> to execute CGI programs.

Note that the specific Windows logon must have read/write access rights to the LANSA installation directory, as a minimum, in order to run LANSA applications properly. (Normally, the default Windows logon does not have enough privileges to access the LANSA installation directory and will not be able to run LANSA applications properly.)

The specific Windows logon is also required for obtaining log files from jsmdirect.exe. For more details refer to [Log files \(Windows\)](#).

Ensure that x_lansa.pro contains an entry USER=<the LANSA user profile that JSM http requests will be given> or PSTC=Y. When a request is received by JSM to run a function, this is the LANSA user profile that the job will run under. Also provide the DBUS and PSWD parameters, unless database trusted connections are being used or the user id and password are contained in the ODBC DSN.

To specify the Windows logon:

(This description is based on the Internet Services Manager for Windows 2000. The equivalent administration programs in other Windows versions may be slightly different.)

1. Start the Internet Services Manager.
2. Browse to the appropriate virtual directory eg. *cgi-bin* and right click on the file *jsmdirect.exe* in the virtual directory and select menu item *Properties*.
3. The *jsmdirect.exe Properties* dialog box will open. Select the *File Security* tab and then the *Edit* button in the frame titled *Anonymous access and authentication control*.
4. The *Authentication Methods* dialog box will open. Check (select) the *Anonymous access* option and press the *Edit* button next to the *Account used*

for anonymous access label.

5. The *Anonymous User Account* dialog box will open. Uncheck (i.e. deselect) the *Allow IIS to control password* option - this is IMPORTANT if you specify a Windows logon username which is not located on the workstation - then specify the Windows logon username and password *of the user profile* used to run LANSA applications. The case of the password must match the case of the password as it was entered into Windows user maintenance. Note that if a workstation belongs to a Windows domain and a Windows User exists on both the Domain and the Local (on the workstation), the Domain User logon takes precedence.
6. Close all the above dialogs by clicking on the *OK* button.

Registering Functions for JSMDirect

On Windows and Linux, the JSMDirect program locates the LANSAX function to be executed by performing a keyed lookup on file dc_w29.txt. On Windows, dc_w29.txt is by default stored in the same folder as jsmdirect.exe. On Linux, dc_w29.txt is in the \$LANSAXROOT/integrator/jsmdirect directory.

The Host Name is the host component of the request URL this is determined by the HTTP protocol Host keyword.

If the URL is `http://www.lansa.com/cgi-bin/jsmdirect` then the Host Name is `WWW.LANSAX.COM:80`.

If the URL is `http://10.1.2.3/cgi-bin/jsmdirect` then the Host Name is `10.1.2.3:80`.

This allows multi-homing where the IP address of `www.lansa.com` and `10.1.2.3` are the same but the HTTP client program specifies the host name in the HTTP protocol allowing the HTTP server to treat `www.lansa.com` and `10.1.2.3` as different hosts.

A Host Name of `*DEFAULT` means any host.

JSMDirect first does a keyed lookup for on the Application Name and Host Name, if an entry does not exist then a second keyed lookup on Application Name and `*DEFAULT` is done.

To update the dc_w29.txt file use LANSAX Integrator Studio or a text editor (Notepad for example).

The dc_w29.txt file is structured as a list of comma separated lines.

Lines starting with a # character are considered to be comments and are not processed.

The values are given by the token position on the line they correspond to the following dc_w29 layout:

- 1 K Application Name identifying the application to be executed. This is the name entered as part of the URL.
- 2 K Host Name Host Name specified either as `*DEFAULT` or `HostName:Port`.
- 3 Process LANSAX process.
- 4 Function LANSAX function to be executed.

- | | | |
|---|-----------|---|
| 5 | Partition | LANSA partition. The process and function must be located in the specified partition. |
| 6 | Language | Execution language for the function. |
| 7 | Program | 3GL program to be executed. If a 3GL program is specified, then the process, function, partition and language fields should be blank. |

Note: The language code is an optional value in the registration of functions in the IBM i DC@W29 but is a required one in dc_w29.txt on Windows and Linux. If you don't register the language code in the IBM i DC@W29, LANSAs will run the server function using the partition's default language. If you don't register the language code in dc_w29.txt, JSMDirect will end with an error.

Example (without Program parameter):

```
#  
# JSMDirect directive file  
#  
# Application Name, Host Name, Process, Function, Partition, Language, Prog  
#  
ORDERENTRY,MYCOMPUTER:80,JSERVICE,JS015,WIZ,ENG  
ORDERENTRY,*DEFAULT,JSERVICE,JS016,WIZ,ENG  
#
```

4.1.5 JSMDirect Set up on Linux

To use JSMDirect, it needs to be installed in a cgi-bin enabled directory under your web server. `$LANSAXROOT/integrator/cgi-bin` contains the `jsmdirect` CGI program you need to make available. It also contains `jsmadmin`, which you will need in order to register functions for JSMDirect through Integrator Studio. The recommended procedure is to direct your web server to find the programs in this location. In Apache, you can add the following lines to the `httpd.conf` file (where `LANSAXROOT` should be replaced by your install directory):

```
ScriptAlias /cgi-bin/jsmdirect LANSAXROOT/integrator/cgi-bin/jsmdirect
```

You will need to restart the web server after adding these lines (usually by running "apachectl restart").

Another way to make the programs available is to copy them to your existing `cgi-bin` directory. You may want to do this if you already have other CGI programs running and want to keep them all in one location.

In either case, you may need to change the files' permissions so that they can be executed by anyone (or at least by the user associated with the HTTP server, such as `apache`). You will also need to make sure that the `jsmdirect` directory in `$LANSAXROOT/integrator` is accessible.

Registering Functions for JSMDirect

On Windows and Linux, the JSMDirect program locates the LANSA function to be executed by performing a keyed lookup on file dc_w29.txt. On Windows, dc_w29.txt is by default stored in the same folder as jsmdirect.exe. On Linux, dc_w29.txt is in the \$LANSAXROOT/integrator/jsmdirect directory.

The Host Name is the host component of the request URL this is determined by the HTTP protocol Host keyword.

If the URL is `http://www.lansa.com/cgi-bin/jsmdirect` then the Host Name is `WWW.LANSA.COM:80`.

If the URL is `http://10.1.2.3/cgi-bin/jsmdirect` then the Host Name is `10.1.2.3:80`.

This allows multi-homing where the IP address of `www.lansa.com` and `10.1.2.3` are the same but the HTTP client program specifies the host name in the HTTP protocol allowing the HTTP server to treat `www.lansa.com` and `10.1.2.3` as different hosts.

A Host Name of `*DEFAULT` means any host.

JSMDirect first does a keyed lookup for on the Application Name and Host Name, if an entry does not exist then a second keyed lookup on Application Name and `*DEFAULT` is done.

To update the dc_w29.txt file use LANSA Integrator Studio or a text editor (Notepad for example).

The dc_w29.txt file is structured as a list of comma separated lines.

Lines starting with a # character are considered to be comments and are not processed.

The values are given by the token position on the line they correspond to the following dc_w29 layout:

- 1 K Application Name identifying the application to be executed. This is the name entered as part of the URL.
- 2 K Host Name Host Name specified either as `*DEFAULT` or `HostName:Port`.
- 3 Process LANSA process.
- 4 Function LANSA function to be executed.

- | | | |
|---|-----------|---|
| 5 | Partition | LANSA partition. The process and function must be located in the specified partition. |
| 6 | Language | Execution language for the function. |
| 7 | Program | 3GL program to be executed. If a 3GL program is specified, then the process, function, partition and language fields should be blank. |

Note: The language code is an optional value in the registration of functions in the IBM i DC@W29 but is a required one in dc_w29.txt on Windows and Linux. If you don't register the language code in the IBM i DC@W29, LANSAs will run the server function using the partition's default language. If you don't register the language code in dc_w29.txt, JSMDirect will end with an error.

Example (without Program parameter):

```
#  
# JSMDirect directive file  
#  
# Application Name, Host Name, Process, Function, Partition, Language, Prog  
#  
ORDERENTRY,MYCOMPUTER:80,JSERVICE,JS015,WIZ,ENG  
ORDERENTRY,*DEFAULT,JSERVICE,JS016,WIZ,ENG  
#
```

4.2 JSMPProxy

JSMPProxy is an HTTP reverse proxy extension to JSMDirect.

The JSMPProxy cgi-bin program resides on a front-end HTTP Web server and transfers data between the HTTP client browser and the backend JSMDirect cgi-bin program.

Using JSMPProxy is very useful as it allows you to define a single entry point and then redirect to a number of different remote servers.

The JSMPProxy program accepts POSTed content that contains the Content-Length HTTP protocol property. The IBMi JSMPProxy program running on the IBMi Apache server can also accept Chunked-Transfer encoded content.

Note: Chunked-Transfer encoding is not supported on the Windows platform.

The JSMPProxy query string can be in the original keyword format or the industry standard name-value pairs. The name service is reserved.

To display the JSMPProxy version number invoke the program with no arguments. An HTML About page will be returned.

`http://mycompany/cgi-bin/jsmpproxy`
`http://mycompany/cgi-bin/jsmpproxy.exe`

Using JSMPProxy, a LANSAs function can be invoked as follows:

`http://mycompany/cgi-bin/jsmpproxy?appname`
`http://mycompany/cgi-bin/jsmpproxy?service=appname`
`http://mycompany/cgi-bin/jsmpproxy.exe?appname`
`http://mycompany/cgi-bin/jsmpproxy.exe?service=appname`

For example, to run the ORDERENTRY application:

`http://mycompany/cgi-bin/jsmpproxy?orderentry`
`http://mycompany/cgi-bin/jsmpproxy?service=orderentry`
`http://mycompany/cgi-bin/jsmpproxy.exe?orderentry`
`http://mycompany/cgi-bin/jsmpproxy.exe?service=orderentry`

The JSMPProxy program locates the LANSAs function to be executed based on the *appname*.

JSMPProxy requires an installed HTTP Server.

JSMPProxy does not require or use any of the LANSAs for the Web software.

Checks are made to make sure that each HTTP protocol header value does not

exceed 4096 bytes.

The following list shows the HTTP protocol header entries forwarded by the proxy program:

- Accept
- Referer
- Cookie
- User-Agent
- SoapAction
- Content-Type
- Content-Length
- AS2-To
- AS2-From
- AS2-Version
- From
- Date
- Subject
- Message-Id
- Disposition-Notification-To
- Disposition-Notification-Options
- Receipt-Delivery-Option

4.2.1 JSMProxy Set up on IBM i

The Apache server can be configured to run CGI programs in %%BINARY/MIXED%%, %%BINARY/BINARY%% or BINARY mode.

The preferred mode is BINARY.

By default, the JSMProxy CGI program runs under the user profile QTMHHTTP1.

Following is an example IBM Apache Server Configuration:

```
# Apache Configuration - JSM Services
Options None
Listen 10.2.0.170:1099
ServerRoot /www/jsmapache
DocumentRoot /www/jsmapache/htdocs
# DefaultFsCCSID 37
# DefaultNetCCSID 819
# ServerUserID USERPROFILE
#
LogLevel Warn
LogCycle Daily
ErrorLog logs/error_log
CustomLog logs/access_log combined
LogFormat "%{User-agent}i" agent
LogFormat "%{Referer}i -> %U" referer
LogFormat "%{Cookie}n \"%r\" %t" cookie
LogFormat "%h %l %u %t \"%r\" %>s %b" common
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-
Agent}i\"" combined
#
SetEnvIf "User-Agent" "Mozilla/2" nokeepalive
SetEnvIf "User-Agent" "JDK/1\0" force-response-1.0
SetEnvIf "User-Agent" "Java/1\0" force-response-1.0
SetEnvIf "User-Agent" "RealPlayer 4\0" force-response-1.0
SetEnvIf "User-Agent" "MSIE 4\0b2;" nokeepalive
SetEnvIf "User-Agent" "MSIE 4\0b2;" force-response-1.0
#
# StartCGI 1
```

```
# SendBufferSize 32768
# ReceiveBufferSize 32768
#
# Use name-based virtual hosting
NameVirtualHost 10.2.0.170
#
<VirtualHost 10.2.0.170>
# The first virtual host directive will become the default host
# This traps the use of the IP address, unsupported or no host name
# It also has no authority to access the document root directory
Options None
ServerName 10.2.0.170
DocumentRoot /www/jsmapache/htdocs
</VirtualHost>
#
<VirtualHost 10.2.0.170>
Options None
ServerName LANSA01
ServerAlias LANSA01.LANSA.COM.AU
DocumentRoot /www/jsmapache/htdocs-site1
CGIConvMode BINARY
ScriptAliasMatch ^/cgi-
bin/jsmpoxy(.*) /qsys.lib/devjms.lib/jsmpoxy.pgm$1
Timeout 3000
#
<Directory /www/jsmapache/htdocs-site1>
Options None
Order Allow,Deny
Allow from all
AllowOverride None
</Directory>
#
<Directory /qsys.lib/devjms.lib>
Options None
Order Allow,Deny
Allow from all
AllowOverride None
# Require valid-user
# AuthType Basic
```

```
# AuthName "Restricted Service"
# UserID QTMHHTTP1
# PasswdFile %%SYSTEM%%
</Directory>
</VirtualHost>
#
# Global server directives
#
<Directory />
Options None
Order Allow,Deny
Deny from all
AllowOverride None
</Directory>
#
<Directory /www/jsmapache/htdocs>
Options None
Order Allow,Deny
Deny from all
AllowOverride None
</Directory>
```

Registering Functions for JSMPProxy

On IBM i the JSMPProxy program performs a keyed lookup on file DC@W30V1 to locate the remote host and program that will receive the forwarded request.

The Host Name is the host component of the request URL this is determined by the HTTP protocol Host keyword.

If the URL is `http://www.lansa.com/cgi-bin/jsmpproxy` then the Host Name is `WWW.LANSA.COM:80`.

If the URL is `http://10.1.2.3/cgi-bin/jsmpproxy` then the Host Name is `10.1.2.3:80`.

This allows multi-homing where the IP address of `www.lansa.com` and `10.1.2.3` are the same but the HTTP client program specifies the host name in the HTTP protocol allowing the HTTP server to treat `www.lansa.com` and `10.1.2.3` as different hosts.

A Host Name of `*DEFAULT` means any host.

JSMPProxy first does a keyed lookup for on the Application Name and Host Name, if an entry does not exist then a second keyed lookup on Application Name and `*DEFAULT` is done.

If the service (application name) does not exist, JSMPProxy returns an error message otherwise the connection is forwarded to the selected remote host and remote program.

To update the DC@W30 file use LANSA Integrator Studio or you may use Data File Utility or SQL commands on the file.

Structure of file DC@W30:

Field	Description	Details
K W30SRV	Application Name	Name identifying the application to be executed on the remote host.
K W30LHO	Host Name	Host Name specified either as <code>*DEFAULT</code> or <code>HostName:Port</code> .
W30MTD	Method Accepted	The proxy program checks the method field to see if the request method is allowed. Possible method field values are: GET POST

GET/POST
*ANY

W30LMT	Content Limit	The proxy program checks the content limit field to see if the inbound content does not exceed the limit value.
W30RHO	Remote Host	Remote Host specified as HostName:Port.
W30PGM	Remote Program	If the remote program is blank, it will default to /cgi-bin/jsmdirect.

Example

```
ORDERENTRY LANSA01:89 GET/POST 100000 LANSA01:88 /cgi-  
bin/jsmdirect  
ORDERENTRY *DEFAULT GET/POST 100000 LANSA01:88 /cgi-  
bin/jsmdirect
```

JSMProxy Program Logic

- Remote client program using HTTP protocol connects to HTTP server.
- HTTP server executes JSMPROXY program. (Look for messages in HTTP server job log).
- JSMProxy program checks for arguments, if no arguments sends an about HTML page to client program.
- JSMProxy program calls exit program JSMPXYEXT (if present) with an EVENT parameter value of ENTRY.
- JSMProxy program uses the service argument value and SERVER_HOST and SERVER_PORT environment variable values to do a keyed lookup on DC@W30V1 to determine the host to receive the request. Also it checks if the HTTP GET or POST protocol and amount of content is allowed for the requested service. If no record is found then another keyed lookup is done using the service argument value and the host value of *DEFAULT. If no record is found then an error message is returned and logged via the JSMPXYEXT program with the EVENT parameter set to an ERRnnnn value. (Look for messages in HTTP server job log).
- JSMProxy program connects to the specified host and forwards selected HTTP protocol properties and standard input.
- JSMProxy program waits for the remote JSMDirect program to complete.
- JSMProxy program writes the HTTP protocol response to standard output.
- JSMProxy program calls exit program JSMPXYEXT (if present) with an EVENT parameter value of EXIT.

4.2.2 JSMPProxy Set up on Windows

If IIS is installed on the target PC, the install/upgrade will automatically configure IIS so that the physical folder JSMCGI is mapped to the virtual folder cgi-bin.

If IIS is not installed, the user will need to manually configure their Web Server.

As in [4.1.4 JSMDirect Set up on Windows](#), you need to specify the Windows logon for executing the jsmproxy.exe CGI program.

Apply the same procedure as in JSMDirect Set up on Windows for the CGI program JSMPProxy.exe in the appropriate virtual directory.

The specific Windows logon is also required for obtaining log files from JSMPProxy.exe.

See [4.9 Log Files \(Windows\)](#) for more information.

Registering Functions for JSMProxy

On Windows and Linux, the JSMProxy program performs a keyed lookup on file dc_w30.txt to locate the remote host and program that will receive the forwarded request.

The Host Name is the host component of the request URL this is determined by the HTTP protocol Host keyword.

If the URL is `http://www.lansa.com/cgi-bin/jsmpoxy` then the Host Name is `WWW.LANSA.COM:80`.

If the URL is `http://10.1.2.3/cgi-bin/jsmpoxy` then the Host Name is `10.1.2.3:80`.

This allows multi-homing where the IP address of `www.lansa.com` and `10.1.2.3` are the same but the HTTP client program specifies the host name in the HTTP protocol allowing the HTTP server to treat `www.lansa.com` and `10.1.2.3` as different hosts.

A Host Name of `*DEFAULT` means any host.

JSMProxy first does a keyed lookup for on the Application Name and Host Name, if an entry does not exist then a second keyed lookup on Application Name and `*DEFAULT` is done.

If the service (application name) does not exist, JSMProxy returns an error message otherwise the connection is forwarded to the selected remote host and remote program.

To update the dc_w30.txt file use LANSA Integrator Studio or a text editor (Notepad for example).

The dc_w30.txt file is structured as a list of comma separated lines.

Lines starting with a # are comments and will not be processed.

Each line in the text file is an individual entry. The fields are separated with a comma.

Structure of file dc_w30.txt:

Field Description Details

No

- | | | |
|---|------------------|---|
| 1 | Application Name | Name identifying the application to be executed on the remote host. |
| 2 | Host Name | Host Name specified either as <code>*DEFAULT</code> or |

HostName:Port.

- | | | |
|---|-----------------|---|
| 3 | Method Accepted | The proxy program checks the method field to see if the request method is allowed. Possible method field values are:
GET
POST
GET/POST
*ANY |
| 4 | Content Limit | The proxy program checks the content limit field to see if the inbound content does not exceed the limit value. |
| 5 | Remote Host | Remote Host specified as HostName:Port. |
| 6 | Remote Program | If the remote program is blank, it will default to /cgi-bin/jsmdirect. |

Example

```
#  
# JSMProxy directive file  
#  
# service, local host, method, limit, remote host, remote program  
#  
ORDERENTRY,LANSA01:89,GET/POST,100000,LANSA01:88,/cgi-  
bin/jsmdirect  
ORDERENTRY,*DEFAULT,GET/POST,100000,LANSA01:88,/cgi-  
bin/jsmdirect  
#
```

4.2.3 JSMProxy Set up on Linux

As with JSMDirect, JSMProxy needs to be installed in a cgi-bin enabled directory under your web server. \$LANSAXROOT/integrator/cgi-bin contains the jsmproxy CGI program you need to make available. It also contains JSMAdmin, which you will need in order to register functions for JSMProxy through Integrator Studio.

The recommended procedure is to direct your web server to find the programs in this location. In Apache, you can add the following lines to the httpd.conf file (where LANSAXROOT should be replaced by your install directory):

```
ScriptAlias /cgi-bin/jsmproxy LANSAXROOT/integrator/cgi-bin/jsmproxy
```

You will need to restart the web server after adding these lines (usually by running "apachectl restart").

Another way to make the programs available is to copy them to your existing cgi-bin directory. You may want to do this if you already have other CGI programs running and want to keep them all in one location.

In either case, you may need to change the files' permissions so that they can be executed by anyone (or at least by the user associated with the HTTP server, such as apache). You will also need to make sure that the jsmproxy directory in \$LANSAXROOT/integrator is accessible.

Registering Functions for JSMPProxy

On Windows and Linux, the JSMPProxy program performs a keyed lookup on file dc_w30.txt to locate the remote host and program that will receive the forwarded request.

The Host Name is the host component of the request URL this is determined by the HTTP protocol Host keyword.

If the URL is http://www.lansa.com/cgi-bin/jsmpproxy then the Host Name is WWW.LANSA.COM:80.

If the URL is http://10.1.2.3/cgi-bin/jsmpproxy then the Host Name is 10.1.2.3:80.

This allows multi-homing where the IP address of www.lansa.com and 10.1.2.3 are the same but the HTTP client program specifies the host name in the HTTP protocol allowing the HTTP server to treat www.lansa.com and 10.1.2.3 as different hosts.

A Host Name of *DEFAULT means any host.

JSMPProxy first does a keyed lookup for on the Application Name and Host Name, if an entry does not exist then a second keyed lookup on Application Name and *DEFAULT is done.

If the service (application name) does not exist, JSMPProxy returns an error message otherwise the connection is forwarded to the selected remote host and remote program.

To update the dc_w30.txt file use LANSA Integrator Studio or a text editor (Notepad for example).

The dc_w30.txt file is structured as a list of comma separated lines.

Lines starting with a # are comments and will not be processed.

Each line in the text file is an individual entry. The fields are separated with a comma.

Structure of file dc_w30.txt:

Field Description Details

No

- | | | |
|---|------------------|---|
| 1 | Application Name | Name identifying the application to be executed on the remote host. |
| 2 | Host Name | Host Name specified either as *DEFAULT or |

HostName:Port.

- | | | |
|---|-----------------|---|
| 3 | Method Accepted | The proxy program checks the method field to see if the request method is allowed. Possible method field values are:
GET
POST
GET/POST
*ANY |
| 4 | Content Limit | The proxy program checks the content limit field to see if the inbound content does not exceed the limit value. |
| 5 | Remote Host | Remote Host specified as HostName:Port. |
| 6 | Remote Program | If the remote program is blank, it will default to /cgi-bin/jsmdirect. |

Example

```
#  
# JSMProxy directive file  
#  
# service, local host, method, limit, remote host, remote program  
#  
ORDERENTRY,LANSA01:89,GET/POST,100000,LANSA01:88,/cgi-  
bin/jsmdirect  
ORDERENTRY,*DEFAULT,GET/POST,100000,LANSA01:88,/cgi-  
bin/jsmdirect  
#
```

4.3 JSMAdmin

The JSMADMIN program allows LANSAs Integrator Studio to maintain the DC@W29, dc_w29.txt, DC@W30 and dc_w30.txt files.

To display the JSMAdmin version number invoke the program with no arguments. An HTML About page will be returned.

<http://mycompany/cgi-bin/jsmadmin>

<http://mycompany/cgi-bin/jsmadmin.exe>

4.3.1 JSMAdmin Set up on IBM i

The Apache server can be configured to run CGI programs in %%BINARY/MIXED%%, %%BINARY/BINARY%% or BINARY mode.

The preferred mode is BINARY.

By default, the JSMAdmin CGI program runs under the user profile QTMHHTTP1.

Following is an example IBM Apache Server Configuration:

```
# Apache Configuration - JSM Services
Options None
Listen 10.2.0.170:1099
ServerRoot /www/jsmapache
DocumentRoot /www/jsmapache/htdocs
# DefaultFsCCSID 37
# DefaultNetCCSID 819
# ServerUserID USERPROFILE
#
LogLevel Warn
LogCycle Daily
ErrorLog logs/error_log
CustomLog logs/access_log combined
LogFormat "%{User-agent}i" agent
LogFormat "%{Referer}i -> %U" referer
LogFormat "%{Cookie}n \"%r\" %t" cookie
LogFormat "%h %l %u %t \"%r\" %>s %b" common
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-
Agent}i\"" combined
#
SetEnvIf "User-Agent" "Mozilla/2" nokeepalive
SetEnvIf "User-Agent" "JDK/1\0" force-response-1.0
SetEnvIf "User-Agent" "Java/1\0" force-response-1.0
SetEnvIf "User-Agent" "RealPlayer 4\0" force-response-1.0
SetEnvIf "User-Agent" "MSIE 4\0b2;" nokeepalive
SetEnvIf "User-Agent" "MSIE 4\0b2;" force-response-1.0
#
# StartCGI 1
```

```
# SendBufferSize 32768
# ReceiveBufferSize 32768
#
# Use name-based virtual hosting
NameVirtualHost 10.2.0.170
#
<VirtualHost 10.2.0.170>
# The first virtual host directive will become the default host
# This traps the use of the IP address, unsupported or no host name
# It also has no authority to access the document root directory
Options None
ServerName 10.2.0.170
DocumentRoot /www/jsmapache/htdocs
</VirtualHost>
#
<VirtualHost 10.2.0.170>
Options None
ServerName LANSA01
ServerAlias LANSA01.LANSA.COM.AU
DocumentRoot /www/jsmapache/htdocs-site1
CGIConvMode BINARY
ScriptAliasMatch ^/cgi-
bin/jsmpoxy(.*) /qsys.lib/devjms.lib/jsmpoxy.pgm$1
ScriptAliasMatch ^/cgi-
bin/jsmdirect(.*) /qsys.lib/devjms.lib/jsmdirect.pgm$1
Timeout 3000
#
<Directory /www/jsmapache/htdocs-site1>
Options None
Order Allow,Deny
Allow from all
AllowOverride None
</Directory>
#
<Directory /qsys.lib/devjms.lib>
Options None
Order Allow,Deny
Allow from all
AllowOverride None
```

```
# Require valid-user
# AuthType Basic
# AuthName "Restricted Service"
# UserID QTMHHTTP1
# PasswdFile %%SYSTEM%%
</Directory>
</VirtualHost>
#
<VirtualHost 10.2.0.170>
Options None
ServerName STUDIOADMIN
DocumentRoot /www/jsmapache/htdocs-site2
CGIConvMode BINARY
ScriptAliasMatch ^/cgi-
bin/jsmadmin(.*?) /qsys.lib/devjsm.lib/jsmadmin.pgm$1
#
<Directory /www/jsmapache/htdocs-site2>
Options None
Order Allow,Deny
Allow from all
AllowOverride None
</Directory>
#
<Directory /qsys.lib/devjsm.lib>
Options None
Order Allow,Deny
Allow from 10.2.1.46
Allow from 10.2.1.47
Allow from 10.2.1.48
AllowOverride None
</Directory>
</VirtualHost>
#
# Global server directives
#
<Directory />
Options None
Order Allow,Deny
Deny from all
```

```
    AllowOverride None
</Directory>
#
<Directory /www/jsmapache/htdocs>
    Options None
    Order Allow,Deny
    Deny from all
    AllowOverride None
</Directory>
```

4.3.2 JSMAAdmin Set up on Windows

If IIS is installed on the target PC, the install/upgrade will automatically configure IIS so that the physical folder JSMCGI is mapped to the virtual folder cgi-bin.

If IIS is not installed, the user will need to manually configure their Web Server. The JSMAAdmin.exe needs to be able to modify the dc_w29.txt and dc_w30.txt files.

As in [4.1.4 JSMDirect Set up on Windows](#), you need to specify the Windows logon for executing the JSMAAdmin.exe CGI program.

Apply the same procedure as in JSMDirect Set up on Windows for the CGI program JSMAAdmin.exe in the appropriate virtual directory.

The specific Windows logon is also required for obtaining log files from JSMAAdmin.exe.

See [4.9 Log Files \(Windows\)](#) for more information.

Changing Permissions

Step 1. Determine the user using the cgi-bin virtual folder

1. Open "Internet Information Services".
2. Find the cgi-bin virtual folder (or the virtual folder you are using).
3. Right mouse click on the cgi-bin virtual folder, and choose Properties.
4. Choose the "Directory Security" tab.
5. Click on "Edit".
6. In the "Anonymous access" group, the "User name:" field is the NT account that is used to run the jobs under cgi-bin folder.

This is the user that needs to have write permissions to the JSMCGI folder.

Remember the name (you will need it in the next step).

7. Close all dialogs and property pages you opened in this step.

Step 2. Change permissions for this user

1. Open Explorer.
2. Navigate to the parent folder of the cgi-bin folder. The cgi-bin folder is

typically C:\Program Files\LANSA\Integrator\JSMCGI, so navigate to C:\Program Files\LANSA\Integrator.

3. Right mouse click on the JSMCGI folder, and choose Properties.
4. Choose the "Security" tab.
5. Click on "Add".
6. Choose the user found in Step 1 (6 above), typically IUSR_<pcname>.
7. Ensure that the "Write" permission is turned on for the user you just added.
8. Click OK to save the new settings.

4.4 Location of the LANSAs System (IBM i)

On IBM i servers, the location of the LANSAs system is determined by the HTTP Servers job's library list. The JSMDRTEXT exit program can be used to modify the library list at execution time. The exit program JSMLASEXT can be used to modify how LANSAs is executed.

4.5 Location of the LANSAX System (Windows)

On Windows servers, the location of the LANSAX system is determined by the LANSAXROOT entry in the jsmdirect.cfg file. The jsmdirect.cfg file is located in the same folder as JSMDirect.exe program.

```
LANSAXROOT=C:\Program Files\LANSAX\X_WIN95
```

This entry was created when JSMDirect was installed.

An existing value is not overwritten by an upgrade or EPC.

If this entry is missing an error message will be logged in jsmdirect.log file.

4.6 Location of the LANSAX System (Linux)

On Linux servers, the location of the LANSAX system is determined by the LANSAXROOT entry in the jsmdirect.cfg file. The entries in the jsmdirect.cfg file are used to define environment variables before the x_run program is executed.

```
LANSAXROOT=/lansa
```

This entry was created when JSMDirect was installed.

If the HTTP Server uses authentication then the x_run parameter USER will be defined following the rules:

- if the env variable USER_NAME is defined then
 - the value of USER_NAME will be passed on to x_run
- else if the env variable REMOTE_USER is defined then
 - the value of REMOTE_USER will be passed on to x_run
- else
 - x_run will be called without defining the parameter USER
 - i.e. The user defined in x_lansa.pro will be used

4.7 Exit Programs (IBM i)

On Windows servers, only information is logged. See [4.9 Log Files \(Windows\)](#).

On IBM i servers, exit programs are called if they are found in the library list of the HTTP Web server instance job.

QGPL or the library that contains the JSMDIRECT.PGM and JSMPROXY.PGM (JSMLIB as specified during the LANSAs Integrator install) are good locations to store these exit programs.

The ENTRY event is a good place to add instructions to modify the job attributes of the server instance, such as adding library entries to the library list. This is necessary if you are invoking LANSAs functions, as the LANSAs libraries must be added to the library list. The shipped JSMDRTEXT adds the default LANSAs libraries DC@PGMLIB and DC@DTALIB to the library list. You will need to modify the user exit if your LANSAs libraries are not these values.

For examples of exit programs, refer to [4.7.1 JSMDRTEXT](#) and [4.7.3 JSMPXYEXT](#).

4.7.1 JSMDRTEXT

The JSMDIRECT program calls CL program JSMDRTEXT when the following life cycle events occur:

ENTRY	Program starts.
EXIT	Program finishes successfully.
ERRnnnn	where nnnn is a 4 digit number starting from 3000. Error has occurred, program ending, EXIT event will not be called

The JSMDRTEXT program reads the JSMDRTDTA data area to get any libraries that need to be added to the current CGI job before the JSMLSAEXT/LANSA program is called. Blank library entries are ignored. By default the JSMDRTDTA data area is blank. During the initial LANSA install, the LANSA program and communication libraries are added to first two positions.

The data area is 2000 bytes in size and the layout is:

1-10	Library (LANSA Program Library)
11-10	Library (LANSA Communication Library)
21-10	Library
xx-10	Libraries...
1991-10	Library

The source code for this exit program is stored in QCLSRC in the JSM library.

```
/* JSMDIRECT EXIT PROGRAM */
```

PGM PARM(&EVENT &SERVICE &SERVERHOST &HOST &PORT &RE

DCL VAR(&EVENT) TYPE(*CHAR) LEN(10)
DCL VAR(&SERVICE) TYPE(*CHAR) LEN(30)
DCL VAR(&SERVERHOST) TYPE(*CHAR) LEN(80)
DCL VAR(&HOST) TYPE(*CHAR) LEN(80)
DCL VAR(&PORT) TYPE(*CHAR) LEN(5)
DCL VAR(&REMOTEUSER) TYPE(*CHAR) LEN(30)
DCL VAR(&REMOTEADDR) TYPE(*CHAR) LEN(45)
DCL VAR(&CONTINUE) TYPE(*CHAR) LEN(1)
DCL VAR(&MESSAGE) TYPE(*CHAR) LEN(256)

DCL VAR(&JOBNAME) TYPE(*CHAR) LEN(10)
DCL VAR(&JOBUSER) TYPE(*CHAR) LEN(10)
DCL VAR(&JOBNUMBER) TYPE(*CHAR) LEN(6)
DCL VAR(&JOBCMD) TYPE(*CHAR) LEN(50)
DCL VAR(&JOBMSG) TYPE(*CHAR) LEN(100)
DCL VAR(&JOBCHGSTS) TYPE(*CHAR) LEN(7) VALUE(OK)

DCL VAR(&TMPLIB) TYPE(*CHAR) LEN(10)
DCL VAR(&TMPLIBPOS) TYPE(*DEC) LEN(5)
DCL VAR(&TMPLIBLIST) TYPE(*CHAR) LEN(2000)

MONMSG MSGID(CPF0000)

/* RETRIEVE LIBRARIES */

RTVDTAARA DTAARA(JSMDRTDTA (1 2000)) RTNVAR(&TMPLIBLIS'

/* CLEANUP ANY PREVIOUS LEFTOVER FAILED SCENARIOS */

CHGVAR VAR(&TMPLIBPOS) VALUE(1)

RMVLIB:

CHGVAR VAR(&TMPLIB) VALUE(%SST(&TMPLIBLIST &TMPLIBPOS
IF (&TMPLIB *EQ ' ') THEN(GOTO ENDRMVLIB)

RMVLIBLE LIB(&TMPLIB)

RCVMSG MSGQ(*PGMQ)

```
CHGVAR VAR(&TMPLIBPOS) VALUE(&TMPLIBPOS + 10)
IF (&TMPLIBPOS *GE 2000) THEN(GOTO ENDRMVLIB)
```

```
GOTO RMVLIB
ENDRMVLIB:
```

```
IF COND(%SUBSTRING(&EVENT 1 3) *EQ 'ERR') THEN(DO)
```

```
/* LOG ERROR EVENT */
```

```
SNDPGMMSG MSG('----- JSMDIRECT ERROR -----')
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&EVENT)
SNDPGMMSG MSGID(&EVENT) MSGF(JSMMMSGF)
```

```
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&SERVICE)
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&SERVICE)
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&REMOVED)
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&REMOVED)
SNDPGMMSG MSG('-----')
```

```
/* CREATE STRING - WRKJOB JOB(464971/QTMHHTTP/JSMINST) */
```

```
RTVJOBA JOB(&JOBNAME) USER(&JOBUSER) NBR(&JOBNUMBER)
```

```
CHGVAR VAR(&JOBCMD) VALUE('WRKJOB JOB(' *TCAT +
    &JOBNUMBER *TCAT '/' *TCAT +
    &JOBUSER *TCAT '/' *TCAT +
    &JOBNAME *TCAT ')')
```

```
CHGVAR VAR(&JOBMSG) VALUE('JSMDirect error, use command' *BCA
    &JOBCMD)
```

```
/* SNDMSG MSG(&JOBMSG) TOUSR(*SYSOPR) */
```

```
GOTO END
```

```
ENDDO
```

IF COND(&EVENT *EQ 'ENTRY') THEN(DO)

```
/* DEFAULT VALUE FOR CONTINUE IS 'Y' */
/* CHGVAR VAR(&CONTINUE) VALUE('N') */
/* CHGVAR VAR(&MESSAGE) VALUE('I do not know you') */
/* GOTO END */
```

```
/* PREPARE JOB FOR CURRENT SERVICE */
```

```
/* CALL PGM(JSMCHGJOB) PARM(&JOBCHGSTS) */
```

IF COND(&JOBCHGSTS *NE 'OK') THEN(DO)

```
  CHGVAR VAR(&JOBMSG) VALUE('Change job exception' *BCAT +
    &JOBCHGSTS)
  SNDPGMMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&JOBM
  ENDDO
```

```
/* ADD LIBRARIES FOR JSMLSAEXT AND LANSA CALL */
```

```
CHGVAR VAR(&TMPLIBPOS) VALUE(1)
```

```
ADDLIB:
```

```
  CHGVAR VAR(&TMPLIB) VALUE(%SST(&TMPLIBLIST &TMPLIBPC
  IF (&TMPLIB *EQ ' ') THEN(GOTO ENDADDLIB)
```

```
  ADDLIBLE LIB(&TMPLIB)
  RCVMSG MSGQ(*PGMQ)
```

```
  CHGVAR VAR(&TMPLIBPOS) VALUE(&TMPLIBPOS + 10)
  IF (&TMPLIBPOS *GE 2000) THEN(GOTO ENDADDLIB)
```

```
  GOTO ADDLIB
ENDADDLIB:
```

```
GOTO END
```

ENDDO

IF COND(&EVENT *EQ 'EXIT') THEN(DO)

/* RESTORE JOB FOR NEXT SERVICE */

GOTO END

ENDDO

END: ENDPGM

4.7.2 JSMLSAEXT

The JSMDIRECT program can only internally execute a LANSA function. If a 3GL program needs to be executed or more flexibility is needed when calling the LANSA program, then the JSMLSAEXT program needs to be created.

The source code for this exit program is stored in QCLSRC in the JSM library.

```
/* JSM LANSA/3GL EXIT PROGRAM */
```

```
PGM PARM(&SERVICE &SERVERHOST &HOST &PORT &REMOTEUSER  
&PROCESS &FUNCTION &PARTITION &LANGUAGE &PROGRAM)
```

```
DCL VAR(&SERVICE) TYPE(*CHAR) LEN(30)  
DCL VAR(&SERVERHOST) TYPE(*CHAR) LEN(80)  
DCL VAR(&HOST) TYPE(*CHAR) LEN(80)  
DCL VAR(&PORT) TYPE(*CHAR) LEN(5)  
DCL VAR(&REMOTEUSER) TYPE(*CHAR) LEN(30)  
DCL VAR(&REMOTEADDR) TYPE(*CHAR) LEN(45)
```

```
DCL VAR(&REQUEST) TYPE(*CHAR) LEN(10) VALUE(RUN)  
DCL VAR(&PROCESS) TYPE(*CHAR) LEN(10)  
DCL VAR(&FUNCTION) TYPE(*CHAR) LEN(10)  
DCL VAR(&PARTITION) TYPE(*CHAR) LEN(3)  
DCL VAR(&LANGUAGE) TYPE(*CHAR) LEN(4)  
DCL VAR(&PROGRAM) TYPE(*CHAR) LEN(32)  
DCL VAR(&RDMLX) TYPE(*CHAR) LEN(1)  
DCL VAR(&CONTINUE) TYPE(*CHAR) LEN(1)  
DCL VAR(&MESSAGE) TYPE(*CHAR) LEN(256)
```

```
DCL VAR(&PARM01) TYPE(*CHAR) LEN(256)  
DCL VAR(&PARM02) TYPE(*CHAR) LEN(256)  
DCL VAR(&PARM03) TYPE(*CHAR) LEN(256)  
DCL VAR(&PARM04) TYPE(*CHAR) LEN(256)  
DCL VAR(&PARM05) TYPE(*CHAR) LEN(256)  
DCL VAR(&PARM06) TYPE(*CHAR) LEN(256)  
DCL VAR(&PARM07) TYPE(*CHAR) LEN(256)  
DCL VAR(&PARM08) TYPE(*CHAR) LEN(256)  
DCL VAR(&PARM09) TYPE(*CHAR) LEN(256)
```

```
DCL VAR(&PARM10) TYPE(*CHAR) LEN(256)
```

```
DCL VAR(&TASKID) TYPE(*CHAR) LEN(8)
```

```
DCL VAR(&PCTYPE) TYPE(*CHAR) LEN(1) VALUE(N)
```

```
DCL VAR(&DEVELOPER) TYPE(*CHAR) LEN(1) VALUE(N)
```

```
DCL VAR(&ALLOWMSG) TYPE(*CHAR) LEN(1) VALUE(N)
```

```
DCL VAR(&PCNAME) TYPE(*CHAR) LEN(10)
```

```
DCL VAR(&DATESRC) TYPE(*CHAR) LEN(1) VALUE(S)
```

```
DCL VAR(&BDEBUG) TYPE(*CHAR) LEN(1) VALUE(N)
```

```
DCL VAR(&BDEBUGDEV) TYPE(*CHAR) LEN(10)
```

```
DCL VAR(&BDEBUGMSG) TYPE(*CHAR) LEN(10)
```

```
DCL VAR(&XRUNADPRM) TYPE(*CHAR) LEN(512)
```

```
MONMSG MSGID(CPF0000)
```

```
/* DEFAULT VALUE FOR CONTINUE IS 'Y' */
```

```
/* CHGVAR VAR(&CONTINUE) VALUE('N') */
```

```
/* CHGVAR VAR(&MESSAGE) VALUE('I do not know you') */
```

```
/* GOTO END */
```

```
IF COND(&PROGRAM *NE ' ') THEN(DO)
```

```
  /* CALL RPG */
```

```
  CALL &PROGRAM
```

```
  GOTO END
```

```
ENDDO
```

```
IF COND(&RDMLX *EQ 'Y') THEN(DO)
```

```
  /* CHANGE REQUEST TO RUN RDMLX */
```

```
  CHGVAR VAR(&REQUEST) VALUE(X_RUN)
```

```
ENDDO
```

```
IF COND(&PROCESS *NE '' *AND &FUNCTION *NE ' ') THEN(DO)
```

```
  IF COND(&REQUEST *EQ 'RUN') THEN(DO)
```

```
    /* CALL LANSA V10 OR V11 */
```

```
    CALL PGM(LANSA) PARM(&REQUEST &PROCESS &FUNCTION +  
      &PARM01 &PARM02 &PARM03 &PARM04 &PARM05 +
```

```
        &PARM06 &PARM07 &PARM08 &PARM09 &PARM10 +
        &PARTITION &LANGUAGE &TASKID &PCTYPE +
        &DEVELOPER &ALLOWMSGS
&PCNAME &DATESRC +
        &BDEBUG &BDEBUGDEV &BDEBUGMSG)
    GOTO END

ENDDO

IF COND(&REQUEST *EQ 'X_RUN') THEN(DO)
    /* CALL LANSА V11 - RDMLX */
    CALL PGM(LANSА) PARM(&REQUEST &PROCESS &FUNCTION +
        &PARM01 &PARM02 &PARM03 &PARM04 &PARM05 +
        &PARM06 &PARM07 &PARM08 &PARM09 &PARM10 +
        &PARTITION &LANGUAGE &TASKID &PCTYPE +
        &DEVELOPER &ALLOWMSGS &PCNAME &DATESRC
        &BDEBUG &BDEBUGDEV &BDEBUGMSG &XRUNAD)
    GOTO END

ENDDO

GOTO END

ENDDO

SNDPGMMSG MSG('No program or function specified for execution')

END: ENDPGM
```

4.7.3 JSMPXYEXT

The JSMPROXY program calls CL program JSMPXYEXT when the following life cycle events occur:

ENTRY Program starts.

EXIT Program finishes successfully.

ERRnnnn where nnnn is a 4 digit number starting from 2000.
Error has occurred, program ending, EXIT event will not be called

The source code for this exit program is stored in QCLSRC in the JSM library.

```
/* JSMPROXY EXIT PROGRAM */
```

```
PGM PARM(&EVENT &SERVICE &SERVERHOST &REMOTEUSER &R  
&DIRECTHOST &HOST &PORT &CONTINUE &MESSAGE)
```

```
DCL VAR(&EVENT) TYPE(*CHAR) LEN(10)  
DCL VAR(&SERVICE) TYPE(*CHAR) LEN(30)  
DCL VAR(&SERVERHOST) TYPE(*CHAR) LEN(80)  
DCL VAR(&REMOTEUSER) TYPE(*CHAR) LEN(30)  
DCL VAR(&REMOTEADDR) TYPE(*CHAR) LEN(45)  
DCL VAR(&DIRECTHOST) TYPE(*CHAR) LEN(80)  
DCL VAR(&HOST) TYPE(*CHAR) LEN(80)  
DCL VAR(&PORT) TYPE(*CHAR) LEN(5)  
DCL VAR(&CONTINUE) TYPE(*CHAR) LEN(1)  
DCL VAR(&MESSAGE) TYPE(*CHAR) LEN(256)
```

```
MONMSG MSGID(CPF0000)
```

```
IF COND(%SUBSTRING(&EVENT 1 3) *EQ 'ERR') THEN(DO)
```

```
/* LOG ERROR EVENT */
```

```
SNDPGMMSG MSG('----- JSMPROXY ERROR -----')
```

```
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&EVENT)
SNDPGMMSG MSGID(&EVENT) MSGF(JSMMMSGF)
```

```
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&SERVI
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&SERVE
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&REMO
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&REMO
SNDPGMMSG MSG('-----')
```

```
GOTO END
```

```
ENDDO
```

```
IF COND(&EVENT *EQ 'ENTRY') THEN(DO)
```

```
/* DEFAULT VALUE FOR CONTINUE IS 'Y' */
/* CHGVAR VAR(&CONTINUE) VALUE('N') */
/* CHGVAR VAR(&MESSAGE) VALUE('I do not know you') */
```

```
GOTO END
```

```
ENDDO
```

```
IF COND(&EVENT *EQ 'EXIT') THEN(DO)
```

```
GOTO END
```

```
ENDDO
```

```
END: ENDPGM
```

4.7.4 JSMADMEXT

The JSMADMIN program calls CL program JSMADMEXT when the following life cycle events occur:

ENTRY Program starts.

EXIT Program finishes successfully.

ERRnnnn where nnnn is a 4 digit number starting from 4000.
Error has occurred, program ending, EXIT event will not be called

The source code for this exit program is stored in QCLSRC in the JSM library.

```
/* JSMADMIN EXIT PROGRAM */
```

```
PGM PARM(&EVENT &SERVICE &SERVERHOST &REMOTEUSER &R
```

```
DCL VAR(&EVENT) TYPE(*CHAR) LEN(10)  
DCL VAR(&SERVICE) TYPE(*CHAR) LEN(30)  
DCL VAR(&SERVERHOST) TYPE(*CHAR) LEN(80)  
DCL VAR(&REMOTEUSER) TYPE(*CHAR) LEN(30)  
DCL VAR(&REMOTEADDR) TYPE(*CHAR) LEN(45)  
DCL VAR(&CONTINUE) TYPE(*CHAR) LEN(1)  
DCL VAR(&MESSAGE) TYPE(*CHAR) LEN(256)
```

```
MONMSG MSGID(CPF0000)
```

```
IF COND(%SUBSTRING(&EVENT 1 3) *EQ 'ERR') THEN(DO)
```

```
/* LOG ERROR EVENT */
```

```
SNDPGMMSG MSG('----- JSMADMIN ERROR -----')  
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&EVENT  
SNDPGMMSG MSGID(&EVENT) MSGF(JSMMMSGF)
```

```
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&SERVI
```

```
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&SERVE
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&REMO'
SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA(&REMO'
SNDPGMMSG MSG('-----')
```

```
GOTO END
```

```
ENDDO
```

```
IF COND(&EVENT *EQ 'ENTRY') THEN(DO)
```

```
/* DEFAULT VALUE FOR CONTINUE IS 'Y' */
```

```
/* CHGVAR VAR(&CONTINUE) VALUE('N') */
```

```
/* CHGVAR VAR(&MESSAGE) VALUE('I do not know you') */
```

```
GOTO END
```

```
ENDDO
```

```
IF COND(&EVENT *EQ 'EXIT') THEN(DO)
```

```
GOTO END
```

```
ENDDO
```

```
END: ENDPGM
```

4.7.5 JSMCHGUSER

The JSMCHGUSER program can be used in the JSMDRTEXT program to change the user profile of the current job.

```
DCL VAR(&STATUS) TYPE(*CHAR) LEN(7)
DCL VAR(&USERID) TYPE(*CHAR) LEN(10)
DCL VAR(&PASSWORD) TYPE(*CHAR) LEN(10)

CALL PGM(JSMCHGUSER) PARM(&USERID &PASSWORD &STATUS)
IF COND(&STATUS *NE OK) THEN(DO)
```

4.7.6 JSMCHGJOB

The JSMCHGJOB program can be used in the JSMDRTEXT program to change the library list of the current job to the library list of the current user profile. The user profile of the HTTP instance can be configured using the UserId %%CLIENT%% entry or by calling the JSMCHGUSER program.

```
DCL VAR(&STATUS) TYPE(*CHAR) LEN(7)
```

```
CALL PGM(JSMCHGJOB) PARM(&STATUS)  
IF COND(&STATUS *NE OK) THEN(DO)
```

4.7.7 JSMGETENV

The JSMGETENV program can be used in the JSMADMEXT, JSMPXYEXT, JSMDRTEXT and JSMLSAEXT program to read CGI environment variables.

The program status code will have one of the following values OK, ERROR or NOVAR.

```
DCL VAR(&ENVNAME) TYPE(*CHAR) LEN(128)
```

```
DCL VAR(&ENVVALUE) TYPE(*CHAR) LEN(1024)
```

```
DCL VAR(&ENVSTS) TYPE(*CHAR) LEN(5)
```

```
CHGVAR VAR(&ENVNAME) VALUE(HTTPS)
```

```
CALL PGM(JSMGETENV) PARM(&ENVNAME &ENVVALUE &ENVSTS)
```

```
IF COND(&ENVSTS *NE 'OK') THEN(DO)
```

```
  GOTO END
```

```
ENDDO
```

4.7.8 JSMTRCENV

The JSMTRCENV program can be used in the JSMADMEXT, JSMPXYEXT, JSMDRTEXT and JSMLSAEXT program to print all available CGI environment variables out to the HTTP instance job log.

The HTTP Apache server stores the QUERY_STRING variable value in ASCII. The JSMGETENV program will convert the QUERY_STRING value from ASCII to EBCDIC.

```
CALL PGM(JSMTRCENV)
```

This is a CGI job for IBM HTTP Server instance JSMAPACHE.

----- Start -----

```
QIBM_USE_DESCRIPTOR_STUDIO=Y
UNIQUE_ID=Az2swAoCAKoAABjpFFMAAAAC
SCRIPT_URL=/cgi-bin/jsmdirect
SCRIPT_URI=http://LANSA01:1099/cgi-bin/jsmdirect
HTTP_HOST=lansa01:1099
CONTENT_TYPE=application/json
HTTP_ACCEPT_ENCODING=gzip, deflate
HTTP_ACCEPT=/*/*
HTTP_REFERER=file://C:\Documents%20and%20Settings\alick.SYD\Desktop
CONTENT_LENGTH=515
HTTP_USER_AGENT=Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0
HTTP_X_PROXY_VERSION=11.4.0
HTTP_X_PROXY_SERVER_NAME=LANSA01
HTTP_X_PROXY_SERVER_PORT=1099
HTTP_X_PROXY_REMOTE_ADDRESS=10.2.1.47
HTTP_ACCEPT_LANGUAGE=en-au
HTTP_CONNECTION=Keep-Alive
HTTP_CACHE_CONTROL=no-cache
PATH=/bin:/usr/bin:/usr/ucb:/usr/bsd:/usr/local/bin
SERVER_SIGNATURE=
SERVER_SOFTWARE=Apache
SERVER_NAME=LANSA01
SERVER_ADDR=10.2.0.170
SERVER_PORT=1099
REMOTE_ADDR=10.2.0.170
```

DOCUMENT_ROOT=/www/jsmapache/htdocs
SERVER_ADMIN=[no address given]
SCRIPT_FILENAME=/QSYS.LIB/DEVJSM.LIB/JSMDIRECT.PGM
DOCUMENT_NAME=/QSYS.LIB/DEVJSM.LIB/JSMDIRECT.PGM
REMOTE_PORT=5094
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.0
REQUEST_METHOD=POST
QUERY_STRING=|Ë?>
REQUEST_URI=/cgi-bin/jsmdirect?json
SCRIPT_NAME=/cgi-bin/jsmdirect
DOCUMENT_URI=/cgi-bin/jsmdirect
RULE_FILE=conf/httpd.conf
CGI_MODE=BINARY
CGI_OUTPUT_MODE=BINARY
IBM_CCSID_VALUE=37
CGI_EBCDIC_CCSID=37
CGI_ASCII_CCSID=819
FSCP=37
NETCP=819
HTTPS=OFF
HTTP_X_DIRECT_VERSION=11.4.0
----- End -----

4.7.9 RUNJSMEXT

The STRJSM command submits the RUNJSM program.

The RUNJSM program will call the CL program RUNJSMEXT if it is found in the library list.

The RUNJSM program calls the CL program RUNJSMEXT when the following life cycle events occur:

- ENTRY Before the Java environment starts.
- REBOOT When the Java environment reboots.
Studio Reboot.
- EXIT When the Java environment ends.
Studio Shutdown or ending of the QJVACMDSRV job.

The source code for this exit program is stored in QCLSRC in the JSM library.

```
PGM PARM(&EVENT &INSTANCE &VERSION &JOB)

DCL VAR(&EVENT) TYPE(*CHAR) LEN(10)
DCL VAR(&INSTANCE) TYPE(*CHAR) LEN(50)
DCL VAR(&VERSION) TYPE(*CHAR) LEN(8)
DCL VAR(&JOB) TYPE(*CHAR) LEN(10)

IF COND(&EVENT *EQ 'ENTRY') THEN(DO)
  /* CHGJOB CCSID(37) FOR NATIVE JDBC DRIVER */
  GOTO END
ENDDO

IF COND(&EVENT *EQ 'REBOOT') THEN(DO)
  GOTO END
ENDDO

IF COND(&EVENT *EQ 'EXIT') THEN(DO)
  GOTO END
ENDDO
```

END: ENDPGM

4.8 Deployment (IBM i)

Client applications can communicate with Java Service Manager's running on other machines.

The JSMCLTDA data area needs to be updated with the remote address of the server or the remote address can be specified on the JSM_OPEN call.

The following tables list the object dependencies for each Integrator feature.

JSMDIRECT

JSMDIRECT	CGI Program
DC@W29	Program Database
DC@W29V1	
DC@W29V2	
JSMMSGF	Message File
JSMDRTEXT	Exit Program
JSMLSAEXT	Exit Program

JSMPROXY

JSMPROXY	CGI Program
DC@W30	Program Database
DC@W30V1	
JSMMSGF	Message File
JSMPTYEXT	Exit Program

LANSA Program -Single-connection

JSMCLTDA	Client Data Area
----------	------------------

JSMMSGF	Message File
BI@P262	JSM_CLOSE BIF
BI@P263	JSM_COMMAND BIF
BI@P264	JSM_OPEN BIF
DCXS882X	Service Program

RPG Program Single-connection

JSMCLDTA	Client Data Area
JSMMSGF	Message File
DCXS882X	Service Program

RPGX Program - Multi-connection

JSMCLDTA	Client Data Area
JSMMSGF	Message File
JSMRPGSRV	Service Program

4.9 Log Files (Windows)

The JSMDirect, JSMProxy and JSMAAdmin CGI programs write log information to the files jsmdirect.log, jsmproxy.log and jsmdadmin.log respectively.

The default folder for these CGI programs is the JSMCGI folder.

By default, these log files are created in the folder from which JSMDirect, JSMProxy or JSMAAdmin programs are started, which is normally the JSMCGI folder.

In order to have the log files created properly, the Windows logon used to execute the JSMDirect, JSMProxy or JSMAAdmin programs must have read/write access rights to the directories where jsmdirect.exe, jsmproxy.exe and jsmdadmin.exe are installed.

The location of these log files is configurable, so they can be created elsewhere.

The name of the configuration file for JSMDirect, JSMProxy and JSMAAdmin is jsmcgi.cfg. This file is not created by the install process.

The configuration file must be located in the same directory from which JSMDirect, JSMProxy or JSMAAdmin programs are started.

The log file directory is specified by the LOGFILELOCATION entry in the jsmcgi.cfg file.

If this entry cannot be read or is empty, the default folder is used.

Example

```
LOGFILELOCATION=c:\temp
```

In this case, the log files are created in the c:\temp directory.

```
LOGFILELOCATION=log
```

In this case, the log files are created in a directory named log, relative to the JSMCGI folder where JSMDirect, JSMProxy and JSMAAdmin are installed.

For more information about specifying the Windows logon for executing the programs, refer to Step 4 of the [Task: Configure the Java Service Manager on Windows](#) in the *Installing LANSA on Windows Guide*.

4.10 Apache Directives

The Apache server contains numerous modules and directives that can be used to enable and disable server features.

For more information refer to:

<http://httpd.apache.org/docs/2.0/mod/>

<http://httpd.apache.org/docs/2.0/mod/directives.html>

4.11 Apache URL Rewriting

The Apache server provides a powerful way to do URL manipulations using Rewrite directives.

For more information refer to:

http://httpd.apache.org/docs/2.0/mod/mod_rewrite.html

<http://httpd.apache.org/docs/2.0/misc/rewriteguide.html>

The following examples illustrate how to use these Rewrite directives.

Example 1

Rewrite URL: `"/myservice.jsp?request=xml&service=neworder"` to `"/cgi-bin/jsmdirect?neworder"`

Sample URL: `http://10.2.0.170:1099/myservice.jsp?request=xml&service=neworder`

If the URL contains a service name value pair then `/cgi-bin/jsmdirect?<value>` is called else `/cgi-bin/jsmdirect?default` is call.

```
RewriteCond %{QUERY_STRING} service\=([^\&]+)
RewriteRule ^/myservice.jsp(.*) /cgi-bin/jsmdirect?%1 [L,PT]
RewriteRule ^/myservice.jsp(.*) /cgi-bin/jsmdirect?default [L,PT]
```

The RewriteCond checks the QUERY_STRING for `service=` and parameterizes any value that is not an `&` character.

If the condition is true it executes the next RewriteRule else this rule is skipped and the other rule is executed.

Note:

Query string is not available to the RewriteRule directive.

It has already been moved to the `query_string` environment variable by the time `mod_rewrite` is activated in a per-directory context.

RewriteCond is only good for the first RewriteRule which follows it.

It does not apply to subsequent RewriteRules, so the `%1` backreference becomes undefined for the second, third and other RewriteRules.

Example 2

Rewrite URL: `"/myservice.jsp?type=2&msgid=AXD&status=NEW"` to `"/cgi-`

```
bin/jsmdirect?myservice+type(2)msgid(AXD)+status(NEW)"
```

```
RewriteRule ^/myservice.jsp /myservice.jsp/{QUERY_STRING} [C]  
RewriteRule ^/myservice.jsp/type=([\&]+)&msgid=([\&]+)&status=  
([\&]+) \  
/cgi-bin/jsmdirect?myservice+type($1)+msgid($2)+status($3) [L,PT]
```

Example 3

Rewrite URL: "/parts/00345" to "/cgi-bin/jsmdirect?orderbind+id(00345)"

Sample URL: http://10.2.0.170:1099/parts/00345

```
RewriteRule ^/parts/(.*) /cgi-bin/jsmdirect?orderbind+id($1) [L,PT]
```

Example 4

Rewrite URL: "/parts/00345/abc" to "/cgi-bin/jsmdirect?
orderbind+id(00345)+item(abc)"

Sample URL: http://10.2.0.170:1099/parts/00345/abc

```
RewriteRule ^/parts/(.*/.*) /cgi-bin/jsmdirect?  
orderbind+id($1)+item($2) [L,PT]
```

Sample Apache configuration file

```
# Apache Configuration - JSM Services  
Options None  
Listen 10.2.0.170:1099  
ServerRoot /www/jsmapache  
DocumentRoot /www/jsmapache/htdocs  
# DefaultFsCCSID 37  
# DefaultNetCCSID 819  
# ServerUserID USERPROFILE  
#  
LogLevel Warn  
LogCycle Daily  
ErrorLog logs/error_log
```

```
CustomLog logs/access_log combined
LogFormat "%{User-agent}i" agent
LogFormat "%{Referer}i -> %U" referer
LogFormat "%{Cookie}n \"%r\" %t" cookie
LogFormat "%h %l %u %t \"%r\" %>s %b" common
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-
Agent}i\"" combined
#
SetEnvIf "User-Agent" "Mozilla/2" nokeepalive
SetEnvIf "User-Agent" "JDK/1\.0" force-response-1.0
SetEnvIf "User-Agent" "Java/1\.0" force-response-1.0
SetEnvIf "User-Agent" "RealPlayer 4\.0" force-response-1.0
SetEnvIf "User-Agent" "MSIE 4\.0b2;" nokeepalive
SetEnvIf "User-Agent" "MSIE 4\.0b2;" force-response-1.0
#
# StartCGI 1
# SendBufferSize 32768
# ReceiveBufferSize 32768
#
# Use name-based virtual hosting
NameVirtualHost 10.2.0.170
#
<VirtualHost 10.2.0.170>
# The first virtual host directive will become the default host
# This traps the use of the IP address, unsupported or no host name
# It also has no authority to access the document root directory
Options None
ServerName 10.2.0.170
DocumentRoot /www/jsmapache/htdocs
</VirtualHost>
#
<VirtualHost 10.2.0.170>
Options None
ServerName LANSA01
ServerAlias LANSA01.LANSA.COM.AU
DocumentRoot /www/jsmapache/htdocs-site1
CGIConvMode BINARY
ErrorDocument 403 /noaccess.html
ErrorDocument 404 /notfound.html
```

```
ScriptAliasMatch ^/cgi-
bin/jsmpoxy(.*) /qsys.lib/devjsm.lib/jsmpoxy.pgm$1
ScriptAliasMatch ^/cgi-
bin/jsmdirect(.*) /qsys.lib/devjsm.lib/jsmdirect.pgm$1
ScriptAliasMatch ^/cgi-
sec/jsmdirect(.*) /qsys.lib/secure.lib/jsmdirect.pgm$1
TimeOut 3000
#
RewriteEngine On
# RewriteLog rewrite-site1.log
# RewriteLogLevel 9
#
RewriteCond %{REQUEST_METHOD} ^TRACE
RewriteRule .* - [L,F]
#
RewriteRule ^/parts/(.*/)(.*) /cgi-sec/jsmdirect?
orderbind+id($1)+item($2) [L,PT]
RewriteRule ^/parts/(.*) /cgi-bin/jsmdirect?orderbind+id($1) [L,PT]
RewriteRule ^/parts /cgi-bin/jsmdirect?orderbind+id(*NONE) [L,PT]
#
RewriteCond %{QUERY_STRING} service\=([^\&]+)
RewriteRule ^/myservice(.*) /cgi-bin/jsmdirect?%1 [L,PT]
RewriteRule ^/myservice(.*) /cgi-bin/jsmdirect?default [L,PT]
#
RewriteCond %{TIME_HOUR}%{TIME_MIN} >0905
RewriteCond %{TIME_HOUR}%{TIME_MIN} <1900
RewriteRule ^/mypage.html /day.html [L]
RewriteRule ^/mypage.html /night.html [L]
#
# RewriteCond %{TIME_HOUR}%{TIME_MIN} <1300 [OR]
# RewriteCond %{TIME_HOUR}%{TIME_MIN} >1500
# RewriteRule ^/cgi-bin/jsmdirect(.*) /noaccess.html [L]
#
# RewriteMap companymap txt:/www/jsmapache/company.map
# RewriteRule ^/company/(.*) ${companymap:$1|http://nocompany.com} [L,
#
# RewriteMap hostmap rnd:/www/jsmapache/randomhost.map
# RewriteRule ^/(.*\.(pdf|gif|jpg)) http://${hostmap:static}/$1 [L,R]
#
```

```
<Directory /www/jsmapache/htdocs-site1>
  Options None
  Order Allow,Deny
  Allow from all
  AllowOverride None
</Directory>
#
<Directory /qsys.lib/devjsm.lib>
  Options None
  Order Allow,Deny
  Allow from all
  AllowOverride None
  # Require valid-user
  # AuthType Basic
  # AuthName "Restricted Service"
  # UserID QTMHHTTP1
  # PasswdFile %%SYSTEM%%
</Directory>
#
<Directory /qsys.lib/secure.lib>
  Options None
  Order Allow,Deny
  Allow from all
  AllowOverride None
  Require valid-user
  AuthType Basic
  AuthName "Restricted Service"
  # UserID QTMHHTTP1
  PasswdFile %%%SYSTEM%%
</Directory>
</VirtualHost>
#
<VirtualHost 10.2.0.170>
  Options None
  ServerName STUDIOADMIN
  DocumentRoot /www/jsmapache/htdocs-site2
  CGIConvMode BINARY
  ScriptAliasMatch ^/cgi-
bin/jsmadmin(.*) /qsys.lib/devjsm.lib/jsmadmin.pgm$1
```

```
#
<Directory /www/jsmapache/htdocs-site2>
  Options None
  Order Allow,Deny
  Allow from all
  AllowOverride None
</Directory>
#
<Directory /qsys.lib/devjsm.lib>
  Options None
  Order Allow,Deny
  Allow from 10.2.1.46
  Allow from 10.2.1.47
  Allow from 10.2.1.48
  AllowOverride None
</Directory>
</VirtualHost>
#
# Global server directives
#
<Directory />
  Options None
  Order Allow,Deny
  Deny from all
  AllowOverride None
</Directory>
#
<Directory /www/jsmapache/htdocs>
  Options None
  Order Allow,Deny
  Deny from all
  AllowOverride None
</Directory>
```

4.12 Apache Reverse Proxy

An Apache server can be configured to work as a reverse-proxy server.

A reverse proxy server appears to the client just like an ordinary web server.

No special configuration on the client is necessary.

The client makes ordinary requests for content in the name-space of the reverse proxy.

The reverse proxy then decides where to send those requests, and returns the content as if it was itself the origin.

A typical usage of a reverse proxy is to provide Internet users access to a server that is behind a firewall.

Reverse proxies can also be used to balance load among several back-end servers, or to provide caching for a slower back-end server.

In addition, reverse proxies can be used simply to bring several servers into the same URL space.

If you require a more flexible reverse-proxy configuration, see the RewriteRule directive with the [P] flag.

For more information refer to:

http://httpd.apache.org/docs/2.0/mod/mod_proxy.html

Following is an example IBM Apache Server Configuration:

```
# Apache Configuration - Reverse Proxy
#
LoadModule proxy_module /QSYS.LIB/QHTTPSVR.LIB/QZSRCORE.SRVF
LoadModule proxy_http_module /QSYS.LIB/QHTTPSVR.LIB/QZSRCORE.S
# LoadModule proxy_ftp_module /QSYS.LIB/QHTTPSVR.LIB/QZSRCORE.
# LoadModule proxy_connect_module /QSYS.LIB/QHTTPSVR.LIB/QZSRCO
#
Options None
Listen 10.2.0.170:2334
ServerRoot /www/jsmpoxy
DocumentRoot /www/jsmpoxy/htdocs
# DefaultFsCCSID 37
# DefaultNetCCSID 819
# ServerUserID USERPROFILE
#
```

```
LogLevel Warn
LogCycle Daily
ErrorLog logs/error_log
CustomLog logs/access_log combined
LogFormat "%{User-agent}i" agent
LogFormat "%{Referer}i -> %U" referer
LogFormat "%{Cookie}n \"%r\" %t" cookie
LogFormat "%h %l %u %t \"%r\" %>s %b" common
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-
Agent}i\"" combined
#
SetEnvIf "User-Agent" "Mozilla/2" nokeepalive
SetEnvIf "User-Agent" "JDK/1\0" force-response-1.0
SetEnvIf "User-Agent" "Java/1\0" force-response-1.0
SetEnvIf "User-Agent" "RealPlayer 4\0" force-response-1.0
SetEnvIf "User-Agent" "MSIE 4.0b2;" nokeepalive
SetEnvIf "User-Agent" "MSIE 4.0b2;" force-response-1.0
#
# Use name-based virtual hosting
NameVirtualHost 10.2.0.170
#
<VirtualHost 10.2.0.170>
# The first virtual host directive will become the default host
Options None
ServerName LANSA01
Timeout 3000
ProxyReverse On
ProxyRequests Off
ProxyTimeout 3000
<Proxy *>
    Order Allow,Deny
    Allow from all
</Proxy>
#
# Pass all requests
# ProxyPass / http://lansa01:1099/
# ProxyPassReverse / http://lansa01:1099/
#
# Pass only CGI requests
```

```
ProxyPass /cgi-bin/ http://lansa01:1099/cgi-bin/
ProxyPassReverse /cgi-bin/ http://lansa01:1099/cgi-bin/
</VirtualHost>
#
<Directory />
Options None
Order Allow,Deny
Deny from all
AllowOverride None
</Directory>
#
<Directory /www/jsmpoxy/htdocs>
Options None
Order Allow,Deny
# Deny from all
Allow from all
AllowOverride None
</Directory>
```

4.13 Apache SSL Support

An Apache server can be configured to be a HTTP SSL server. Following is an example IBM Apache Server Configuration:

```
# Apache Configuration - SSL JSM Services
#
LoadModule ibm_ssl_module /QSYS.LIB/QHTTPSVR.LIB/QZSRVSSL.SRV
#
Options None
Listen 10.2.0.170:443
ServerRoot /www/jsmapache
DocumentRoot /www/jsmapache/htdocs
# DefaultFsCCSID 37
# DefaultNetCCSID 819
# ServerUserID USERPROFILE
#
LogLevel Warn
LogCycle Daily
ErrorLog logs/error_log
CustomLog logs/access_log combined
LogFormat "%{User-agent}i" agent
LogFormat "%{Referer}i -> %U" referer
LogFormat "%{Cookie}n \"%r\" %t" cookie
LogFormat "%h %l %u %t \"%r\" %>s %b" common
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-
Agent}i\"" combined
#
SetEnvIf "User-Agent" "Mozilla/2" nokeepalive
SetEnvIf "User-Agent" "JDK/1\0" force-response-1.0
SetEnvIf "User-Agent" "Java/1\0" force-response-1.0
SetEnvIf "User-Agent" "RealPlayer 4\0" force-response-1.0
SetEnvIf "User-Agent" "MSIE 4\0b2;" nokeepalive
SetEnvIf "User-Agent" "MSIE 4\0b2;" force-response-1.0
SetEnvIf "User-Agent" ".*MSIE.*" nokeepalive ssl-unclean-
shutdown downgrade-1.0 force-response-1.0
#
ScriptAliasMatch ^/cgi-bin/jsmadmin(.*) /qsys.lib/jsm.lib/jsmadmin.pgm$1
```

```
ScriptAliasMatch ^/cgi-bin/jsmpoxy(.*) /qsys.lib/jsm.lib/jsmpoxy.pgm$1
ScriptAliasMatch ^/cgi-bin/jsmdirect(.*) /qsys.lib/jsm.lib/jsmdirect.pgm$1
#
# StartCGI 1
# SendBufferSize 32768
# ReceiveBufferSize 32768
#
# Use name-based virtual hosting
NameVirtualHost 10.2.0.170
#
<VirtualHost 10.2.0.170>
# The first virtual host directive will become the default host
Options None
ServerName LANSA01
SSLEngine On
# SSLClientAuth required
SSLAppName QIBM_HTTP_SERVER_JSMSSL
CGIConvMode BINARY
TimeOut 3000
</VirtualHost>
#
<Directory />
Options None
Order Allow,Deny
Deny from all
AllowOverride None
</Directory>
#
<Directory /www/jsmapache/htdocs>
Options None
Order Allow,Deny
Allow from all
AllowOverride None
</Directory>
#
<Directory /qsys.lib/jsmlib.lib>
Options None
Order Allow,Deny
Allow from all
```

AllowOverride None
</Directory>

4.14 Apache Tracing (IBM i)

Use the WRKSPLF SELECT (QTMHHTTP) command to view current spool files owned by this user.

Delete spool files that are not important so your next selection will be easier to manage.

Start the HTTP server instance with the **-vv** option.

```
STRTCPSVR SERVER (*HTTP) HTTPSVR (JSMAPACHE '-vv')
```

Look for completion message CPCA984 in the server job log for confirmation that the trace option you specified has been accepted.

```
User Trace option changed for job
```

```
534123/QTMHHTTP/JSMAPACHE.
```

```
This is the manager job for HTTP Server instance JSMAPACHE.
```

Run the HTTP client program that is causing the error to occur in the HTTP instance.

End the HTTP server instance:

```
ENDTCPSVR SERVER (*HTTP) HTTPSVR (JSMAPACHE)
```

Use the WRKSPLF SELECT (QTMHHTTP) again to view the created trace spool files:

```
QZSRHTTPTR QTMHHTTP PRT01 QSRV534123 HLD 16
```

```
QZSRHTTPTR QTMHHTTP PRT01 QSRV534124 HLD 1
```

```
QZSRHTTPTR QTMHHTTP PRT01 QSRV534125 HLD 30
```

```
QZSRHTTPTR QTMHHTTP PRT01 QSRV534126 HLD 21
```

Also you can start tracing any time (without stopping and starting the instance) by using the following commands:

```
TRCTCPAPP APP(*HTTP) HTTPSVR(JSMAPACHE) TRCLVL(*VERBOSE)
```

```
TRCTCPAPP APP(*HTTP) SET(*OFF)
```

4.15 SSL Support

LANSA Integrator supplied HTTP services can communicate using [SSL](#) via the standard Java SSL extensions.

Use the IBM Information Center site to configure your IBM i for SSL support. Following is an example of JSM manager.properties file:

```
#
# Java Service Manager configuration
#
# javax.net.ssl.keyStore=
# javax.net.ssl.keyStoreType=jks
# javax.net.ssl.keyStorePassword=
# javax.net.ssl.trustStore=
# javax.net.ssl.trustStoreType=jks
# javax.net.ssl.trustStorePassword=
# javax.net.debug=all
# javax.net.debug=ssl,handshake,data,trustmanager
#
# ssl.KeyManagerFactory.algorithm=IBMX509
# ssl.TrustManagerFactory.algorithm=IBMX509
# ssl.SocketFactory.provider=com.ibm.jsse.JSSESocketFactory
# ssl.ServerSocketFactory.provider=com.ibm.jsse.JSSEServerSocketFactory
```

Java Trust/Key Store

By default, IBM Technology for Java JDK's use the cacerts trust/key store file. The location of the cacerts file depends upon the JDK version and bit mode. The cacerts file is located in one of the following directories.

```
/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/security
/QOpenSys/QIBM/ProdData/JavaVM/jdk50/64bit/jre/lib/security
/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit/jre/lib/security
/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit/jre/lib/security
/QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit/jre/lib/security
/QOpenSys/QIBM/ProdData/JavaVM/jdk70/64bit/jre/lib/security
```

By default, IBM Classic JDK's 1.5 and 1.6 use the cacerts trust/key store file. The location of the cacerts file depends upon the JDK version.

```
/QIBM/ProdData/Java400/jdk15/lib/security  
/QIBM/ProdData/Java400/jdk6/lib/security
```

By default, IBM Classic JDK 1.4 uses Digital Certificate Manager. The IBM Classic JDK 1.4 cacerts file is located in the following directory.

```
/QIBM/ProdData/Java400/jdk14/lib/security
```

To configure the IBM Classic JDK 1.4 to use a cacerts file as the trust/key store, the following steps need to be performed.

Edit the manager.properties file and enable the following properties.

```
ssl.KeyManagerFactory.algorithm=IBMX509  
ssl.TrustManagerFactory.algorithm=IBMX509  
ssl.SocketFactory.provider=com.ibm.jsse.JSSESocketFactory  
ssl.ServerSocketFactory.provider=com.ibm.jsse.JSSEServerSocketFactory
```

To select a digital certificate to use for client authentication connections when using Digital Certificate Manager you can use the default certificate or specify your own.

To specify which digital certificate to use, use the following properties:

```
os400.certificateLabel=MYLABEL  
os400.certificateContainer=/QIBM/USERDATA/ICSS/CERT/SERVER/DEFA
```

Digital certificate containers store digital certificates. If you want to use the IBM i system default certificate container, you do not need to specify a certificate container.

To use a specific digital certificate container, you need to specify that digital certificate container.

SSL Client Authentication on the Client

If the server is configured for client authentication then after the client has

authenticated the server, the server requests the client's certificate.

The client then sends its signed certificate, and the server performs the same authentication process as the client did, comparing the client certificate to a library of existing certificates.

If the trust manager is Digital Certificate Manager then the certificate to be sent to the server for authentication is specified by the `os400.certificateLabel` property.

Some servers require the certificate to contain the extended key attribute 'client authentication' (1.3.6.1.5.5.7.3.2).

The CA certificate that signed the client certificate will need to be sent to the other party to be included in the server's list of trusted certificates.

SSL Client Authentication on the Server

If the server is configured for client authentication then after the client has authenticated the server, the server requests the client's certificate.

```
<VirtualHost 10.2.0.170>  
  Options None  
  ServerName LANSA01  
  SSLEngine On  
  SSLClientAuth required  
  SSLAppName QIBM_HTTP_SERVER_JSMSSL  
</VirtualHost>
```

The client then sends its signed certificate, and the server performs the same authentication process as the client did, comparing the client certificate to a library of existing certificates.

4.16 SSL Handshake

The Secure Sockets Layer (SSL) protocol uses a combination of public-key and symmetric-key encryption.

Symmetric-key encryption is much faster than public-key encryption; however, public-key encryption provides better authentication techniques.

An SSL session always begins with an exchange of messages called the SSL handshake.

The handshake allows the server to authenticate itself to the client by using public-key techniques, and then allows the client and the server to cooperate in the creation of symmetric keys used for rapid encryption, decryption, and tamper detection during the session that follows.

Optionally, the handshake also allows the client to authenticate itself to the server.

The steps involved in the SSL handshake are as follows:

1. The client sends a client "hello" message that lists the cryptographic capabilities of the client (sorted in client preference order), such as the version of SSL, the cipher suites supported by the client, and the data compression methods supported by the client. The message also contains a 28-byte random number.
2. The server responds with a server "hello" message that contains the cryptographic method (cipher suite) and the data compression method selected by the server, the session ID, and another random number.

Note: The client and the server must support at least one common cipher suite, or else the handshake fails. The server generally chooses the strongest common cipher suite.

3. The server sends its digital certificate. If the server uses SSL V3, and if the server application requires a digital certificate for client authentication, the server sends a "digital certificate request" message. In the "digital certificate request" message, the server sends a list of the types of digital certificates supported and the distinguished names of acceptable certificate authorities.
4. The server sends a server "hello done" message and waits for a client response.
5. Upon receipt of the server "hello done" message, the client verifies the validity of the server's digital certificate and checks that the server's "hello"

parameters are acceptable.

If the server requested a client digital certificate the client sends a digital certificate, or if no suitable digital certificate is available, the client sends a "no digital certificate" alert. This alert is only a warning, but the server application can fail the session if client authentication is mandatory.

6. The client sends a "client key exchange" message. This message contains the pre-master secret, a 46-byte random number used in the generation of the symmetric encryption keys and the message authentication code (MAC) keys, encrypted with the public key of the server.

If the client sent a digital certificate to the server, the client sends a "digital certificate verify" message signed with the client's private key. By verifying the signature of this message, the server can explicitly verify the ownership of the client digital certificate.

Note: An additional process to verify the server digital certificate is not necessary. If the server does not have the private key that belongs to the digital certificate, it cannot decrypt the pre-master secret and create the correct keys for the symmetric encryption algorithm, and the handshake fails.

7. The client uses a series of cryptographic operations to convert the pre-master secret into a master secret, from which all key material required for encryption and message authentication is derived. Then the client sends a "change cipher spec" message to make the server switch to the newly negotiated cipher suite. The next message sent by the client (the "finished" message) is the first message encrypted with this cipher method and keys.
8. The server responds with a "change cipher spec" and a "finished" message of its own.
9. The SSL handshake ends, and encrypted application data can be sent.

4.17 JSM HTTP Server

JSM includes a HTTP server that can be used to serve static file resources or dynamic content via custom Java classes.

The HTTP server is configured by an XML file and `manager.properties` entries. The XML configuration file is specified by the `httpd` property.

manager.properties

```
httpd=system/httpd.xml
```

The HTTP server configuration file can describe multiple HTTP instances and multiple virtual hosts with an instance. The HTTP instance specifies the TCP/IP communication options and access and error log file locations. IP address access control for allow and deny can be specified at the instance level. Instance wide MIME file type mapping can also be specified. The default value for the port attribute is 80 and the default value for the interface is `*ALL`. The backlog default value is 256.

With a HTTP instance, multiple virtual host sections can be specified. When a HTTP request is read, the HTTP Host property is used to locate the virtual host section to be used to process the request. The virtual host name is case insensitive compared to the Host property name. If a port component exists on the Host property then this is removed before it is used to find a matching virtual host name. If no virtual host section is found then the default `*` virtual host is used. A virtual host element can also contain a root and index attribute to override the instance root and index values. If an instance or virtual elements have an attribute active value of false, then the element is ignored.

IP address access control for allow and deny can be specified at the virtual host level. Also user agent and content length access control can be specified. MIME file type mapping can also be specified.

Basic authentication realms can be setup to restrict access to various resource locations. The virtual host protect element is used to specify a restricted resource.

Java classes can be executed based on various resource paths. The virtual host script element is used to associate a Java classes to a particular resource path. The trace attribute enables JSM tracing of the HTTP transaction. The `clienttrace` attribute allows tracing to be enabled from the browser URL using the query

string URL parameter trace=true.

httpd.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>

  <instance name="WebServer" active="true" root="www/instance/htdocs" ind

    <errorlog enabled="true" file="www/instance/logs/error.log"/>

    <accesslog enabled="true" file="www/instance/logs/access.log"/>

    <listen port="4563" sslport="4564" interface="*ALL" backlog="256"
      secure="false" store="pki/wwwssl.jks" password="password"
      buffersend="-1" bufferreceive="-1" nodelay="false" timeout="5"/>

  <access>

    <!--
      Once a true condition occurs no more evaluations are done.

      <deny address="*" />
      <deny address="10.2.1.45" />

      <allow address="*" />
      <allow address="10.2.1.45" />
    -->

  </access>

  <mimetypes>

    <!--
      These are the default values.
    -->
```

```
<map extension="png" type="image/png"/>
<map extension="gif" type="image/gif"/>
<map extension="jpg" type="image/jpeg"/>
<map extension="jpeg" type="image/jpeg"/>
<map extension="tiff" type="image/tiff"/>
<map extension="ico" type="image/x-icon"/>
<map extension="svg" type="image/svg+xml"/>
<map extension="pdf" type="application/pdf"/>
<map extension="css" type="text/css; charset=utf-8"/>
<map extension="xsl" type="text/xsl; charset=utf-8"/>
<map extension="xml" type="text/xml; charset=utf-8"/>
<map extension="htm" type="text/html; charset=utf-8"/>
<map extension="html" type="text/html; charset=utf-8"/>
<map extension="js" type="application/x-javascript; charset=utf-8"/>
```

```
</mimetype>
```

```
<virtual host="*" active="true">
```

```
<access>
```

```
<!--
```

Once a true condition occurs no more evaluations are done.

```
<deny address="*" />
<deny address="10.2.1" />
<deny address="10.2.1.45" />

<allow address="*" />
<allow address="10.2.1" />
<allow address="10.2.1.45" />

<deny useragent="*" />
<deny useragent="?" />
<deny useragent="webos" />
<deny useragent="opera" />
<deny useragent="chrome" />
<deny useragent="safari" />
<deny useragent="android" />
```

```
<deny useragent="firefox"/>
<deny useragent="explorer"/>
<deny useragent="imac"/>
<deny useragent="ipad"/>
<deny useragent="ipod"/>
<deny useragent="iphone"/>
<deny useragent="iwork"/>
<deny useragent="msnbot"/>
<deny useragent="lansaua"/>
<deny useragent="yahoobot"/>
<deny useragent="googlebot"/>
<deny useragent="googletoolbar"/>
<deny useragent="longreach"/>
<deny useragent="webdavnav"/>
```

```
<allow useragent="*"/>
<allow useragent="?"/>
<allow useragent="webos"/>
<allow useragent="opera"/>
<allow useragent="chrome"/>
<allow useragent="safari"/>
<allow useragent="android"/>
<allow useragent="firefox"/>
<allow useragent="explorer"/>
<allow useragent="imac"/>
<allow useragent="ipad"/>
<allow useragent="ipod"/>
<allow useragent="iphone"/>
<allow useragent="iwork"/>
<allow useragent="msnbot"/>
<allow useragent="lansaua"/>
<allow useragent="yahoobot"/>
<allow useragent="googlebot"/>
<allow useragent="googletoolbar"/>
<allow useragent="longreach"/>
<allow useragent="webdavnav"/>
```

```
<deny contentlength="4096"/> Deny access if content length is greater than 4096
<allow contentlength="4096"/> Allow access if content length less than 4096
```

```
    Zero content length from the browser is a special case and access is al
-->
```

```
<!--
    The default is to allow access for all addresses, useragents and content
-->
```

```
</access>
```

```
<protect>
```

```
  <realm name="Area 51">
```

```
    <!-- access is a hash of user, password and realm -->
```

```
    <user name="user" access="bb644a9819425bfd8586b408896a1031"/>
```

```
  </realm>
```

```
  <match uri="/restricted" realm="Area 51" authentication="basic,digest"/>
```

```
</protect>
```

```
<script>
```

```
  <match uri="/ping.jsp" class="com.lansa.jsm.JSMHTTPServicePing" trac
```

```
  <match uri="/" class="com.lansa.jsm.JSMHTTPServiceFile" trace="false
```

```
    <parameter name="cache.maxage" value="28800"/>
```

```
    <parameter name="cache.maxage.pdf" value="28800"/>
```

```
    <parameter name="cache.maxage.image" value="28800"/>
```

```
  </match>
```

```
</script>
```

```
<mimetype>
```

```
  <map extension="pdf" type="application/pdf"/>
```

```
<!--
```

```
  Defaults to instance mimetype
```

```

-->

</mimetype>

</virtual>

</instance>

</configuration>

```

To write a custom Java classes to process HTTP requests from the HTTP server requires the Java class to implement the com.lansa.jsm.JSMHTTPService interface.

JSMHTTPService interface

```

public interface JSMHTTPService
{
    public void doRequest ( JSMTrace trace,
                          JSMHTTPVirtual virtual,
                          JSMHTTPContext context,
                          JSMHTTPTransport transport,
                          JSMHTTPRequest request ) ;
}

```

JSMHTTPVirtual public methods

```

String getHost ()
boolean isActive () ;
File getDocumentRoot ()
File getDocumentIndex ()
File getFile ( String path )
String getContentType ( File file )
void logException ( JSMHTTPTransport transport, Throwable t )
void logError ( JSMHTTPTransport transport, JSMHTTPRequest request, Stri

```

JSMHTTPContext public methods

HashMap getServiceParameters ()
JSMHTTPHost[] getServiceHosts ()

JSMHTTPHost public methods

String getName ()
HashMap getParameters ()

JSMHTTPTransport public methods

int getId ()
Socket getSocket ()
boolean isSecure ()
String getClientAddress ()
InetAddress getInetAddress ()
InputStream getInputStream ()
OutputStream getOutputStream ()
void consumeInputStream (long length)
byte[] readInputStream (int length)
void sendNotFound (String message)
void sendForbidden (String message)
void sendNotImplemented (String message)

JSMHTTPRequest public methods

String getHead ()
String getMethod ()
String getVersion ()
String getResourceRaw ()
String getResourcePath ()
Properties getProperties ()
String getProperty (String key)
String Enumeration getPropertyNames ()
Properties getQueryParameters ()
String getHost ()
long getContentLength ()
boolean canAcceptGZIP ()

```
String getUserAgent ()
String getUserAgentVersion ()
boolean isUserAgent ( String agent )
boolean isUserAgentIE6 ()
```

The following Java class is the JSM HTTP server class that handles static file requests.

Example

```
package com.acme.service ;

import java.io.* ;

import java.util.Date ;
import java.util.HashMap ;

import java.util.zip.GZIPInputStream ;

import com.lansa.jsm.* ;

public final class Example implements JSMHTTPService
{
    private final static String CRLF = "\r\n" ;
    private final static String EMPTY_STRING = "" ;
    private final static String ENCODING_UTF8 = "UTF-8" ;

    private JSMTrace m_trace = null ;
    private JSMHTTPRequest m_request = null ;
    private JSMHTTPVirtual m_virtual = null ;
    private JSMHTTPTransport m_transport = null ;

    /*
       RFC2616 - Hypertext Transfer Protocol - HTTP/1.1
    */

    private HashMap m_serviceParameters = null ; // Not synchronized
```

```

public Example ()
{
}

public final void doRequest ( JSMTrace trace,
                             JSMHTTPVirtual virtual,
                             JSMHTTPContext context,
                             JSMHTTPTransport transport,
                             JSMHTTPRequest request )
{
    try
    {
        m_trace = trace ;

        m_virtual = virtual ;

        m_request = request ;

        m_transport = transport ;

        m_serviceParameters = context.getServiceParameters () ;

        handleRequest () ;
    }
    catch ( Throwable t )
    {
        /*
         * Log exception
         */

        m_virtual.logException ( m_transport, t ) ;

        if ( m_trace == null )
        {
            System.out.println ( "JSMHTTPServiceFile: handle request exceptio

            t.printStackTrace () ;
        }
        else

```

```

        {
            m_trace.print ( t ) ;
        }
    }
}

```

```
private final void handleRequest () throws IOException
```

```

{
    if ( m_trace != null )
    {
        m_trace.println ( "Handle request for resource path: ", m_request.getRe
    }
}

```

```
/*
```

No request content is expected for GET and HEAD methods

Somebody might have used a POST and content

Need to consume any content on the socket input stream

```

    This allows the browser to switch over and read the HTTP response
*/

```

```
m_transport.consumeInputStream ( m_request.getContentLength () ) ;
```

```
/*
```

Check method

```
*/
```

```
if ( !isAllowedMethod ( m_request.getMethod () ) )
```

```

{
    m_virtual.logError ( m_transport, m_request, "Method is not implemen

```

```

        m_transport.sendNotImplemented ( m_request.getMethod () ) ;

```

```

        return ;

```

```

    }

```

```
/*
```

```
    Get file
    */

    String path = m_request.getResourcePath () ;

    File file = m_virtual.getFile ( path ) ;

    if ( file == null )
    {
        if ( m_trace != null )
        {
            m_trace.println ( "File not found" ) ;
        }

        m_virtual.logError ( m_transport, m_request, "File not found" ) ;

        m_transport.sendNotFound ( path ) ;

        return ;
    }

    /*
    File found
    */

    if ( file.isDirectory () )
    {
        if ( m_trace != null )
        {
            m_trace.println ( "File is a directory: ", file.getAbsolutePath () ) ;
        }

        m_virtual.logError ( m_transport, m_request, "File is a directory" ) ;

        m_transport.sendNotFound ( path ) ;

        return ;
    }
}
```

```

if ( m_request.getMethod().equals ( "HEAD" ) )
{
    /*
        HEAD file
    */

    sendHEAD ( file ) ;

    return ;
}

/*
    GET file
*/

sendFile ( file ) ;

/*
    Remove one-shot directory file
*/

if ( file.getParentFile().getName().equals ( "one-shot" ) )
{
    if ( !file.delete () )
    {
        if ( m_trace != null )
        {
            m_trace.println ( "Cannot delete one-
shot file: ", file.getAbsolutePath () ) ;
        }

        m_virtual.logError ( m_transport, m_request, "Cannot delete one-
shot file" ) ;
    }
}

}

private final void sendHEAD ( File sendFile )

```

```

{
    /*
        The HEAD response is the same as the GET response, except no content
        The Content-Length of the file is included, but no content
    */

    try
    {
        if ( m_trace != null )
        {
            m_trace.println ( "Send HEAD response: ", sendFile.getCanonicalPa
        }

        /*
            Create protocol

            RFC2616 - HTTP/1.1

            If the server chooses to close the connection immediately after sending
            it should send a Connection header including the close token
        */

        long contentLength = sendFile.length () ;

        boolean isCompressed = JSMHTTPHelper.isCompressed ( sendFile ) ;

        if ( isCompressed && !canAcceptCompressed () )
        {
            isCompressed = false ;

            contentLength = JSMHTTPHelper.getUncompressedContentLength
        }

        String contentType = m_virtual.getContentType ( sendFile ) ;

        StringBuffer response = new StringBuffer ( 512 ) ;

        response.append ( "HTTP/1.1 200 OK" ) ;

```

```
response.append ( CRLF ) ;

response.append ( "Date: " ) ;
response.append ( JSMDatetime.getFormattedHTTPDate ( new Date (
response.append ( CRLF ) ;

response.append ( "Content-Type: " ) ;
response.append ( contentType ) ;
response.append ( CRLF ) ;

response.append ( "Content-Length: " ) ;
response.append ( Long.toString ( contentLength ) ) ;
response.append ( CRLF ) ;

if ( isCompressed )
{
    response.append ( "Content-Encoding: gzip" ) ;
    response.append ( CRLF ) ;
}

/*
    Response date is mandatory for caching to work
*/

int cacheAge = getCacheAge ( contentType, sendFile ) ;

if ( cacheAge <= 0 )
{
    response.append ( "Cache-Control: max-age=0, s-maxage=0, must-
revalidate, proxy-revalidate, no-cache" ) ;
    response.append ( CRLF ) ;
}

if ( JSMHTTPHelper.isTextPlain ( contentType ) )
{
    /*
        Stop IE8 from doing content sniffing
    */
```

```

        response.append ( "X-Content-Type-Options: nosniff" );
        response.append ( CRLF );
    }

    response.append ( "Connection: close" );
    response.append ( CRLF );

    response.append ( CRLF );

    byte[] protocol = response.toString().getBytes ( ENCODING_UTF8 );

    if ( m_trace != null )
    {
        File file = m_trace.createTraceFile ( "HTTP_PROTOCOL_RESPON

        JSMHTTPHelper.outputToFile ( file, protocol );
    }

    /*
        Send response

        If the client socket is closed, then a broken pipe exception will occur
    */

    OutputStream outputStream = m_transport.getOutputStream ();

    outputStream.write ( protocol );

    outputStream.flush ();
}
catch ( IOException e )
{
    /*
        The user can close the browser, before all the content is sent
    */

    if ( m_trace != null )
    {
        m_trace.println ( "Error sending HEAD response" );
    }
}

```

```

    }

    m_virtual.logError ( m_transport, m_request, "Error sending HEAD res
}
}

private final void sendFile ( File sendFile )
{
    InputStream inputStream = null ;

    try
    {
        if ( m_trace != null )
        {
            m_trace.println ( "Send file response: ", sendFile.getCanonicalPath (
        }

        /*
        Create protocol

        RFC2616 - HTTP/1.1

        If the server chooses to close the connection immediately after sending
        it should send a Connection header including the close token
        */

        boolean sendChunked = false ;

        boolean uncompressContent = false ;

        long contentLength = sendFile.length () ;

        boolean isCompressed = JSMHTTPHelper.isCompressed ( sendFile ) ;

        if ( isCompressed && !canAcceptCompressed () )
        {
            isCompressed = false ;

            uncompressContent = true ;

```

```

        contentLength = JSMHTTPHelper.getUncompressedContentLength
    }

String contentType = m_virtual.getContentType ( sendFile );

StringBuffer response = new StringBuffer ( 512 );

response.append ( "HTTP/1.1 200 OK" );
response.append ( CRLF );

response.append ( "Date: " );
response.append ( JSMDatetime.getFormattedHTTPDate ( new Date (
response.append ( CRLF );

response.append ( "Content-Type: " );
response.append ( contentType );
response.append ( CRLF );

if ( sendChunked )
{
    response.append ( "Transfer-Encoding: chunked" );
    response.append ( CRLF );
}
else
{
    response.append ( "Content-Length: " );
    response.append ( Long.toString ( contentLength ) );
    response.append ( CRLF );
}

if ( isCompressed )
{
    response.append ( "Content-Encoding: gzip" );
    response.append ( CRLF );
}

```

```

/*

```

Response Date: is mandatory for caching to work

```

*/

int cacheAge = getCacheAge ( contentType, sendFile );

if ( cacheAge <= 0 )
{
    response.append ( "Cache-Control: max-age=0, s-maxage=0, must-
revalidate, proxy-revalidate, no-cache" );
    response.append ( CRLF );
}
else
{
    response.append ( "Cache-Control: " );
    response.append ( "max-age=" );
    response.append ( Integer.toString ( cacheAge ) );
    response.append ( ", s-maxage=" );
    response.append ( Integer.toString ( cacheAge ) );
    response.append ( CRLF );
}

if ( JSMHTTPHelper.isTextPlain ( contentType ) )
{
    /*
    Stop IE8 from doing content sniffing
    */

    response.append ( "X-Content-Type-Options: nosniff" );
    response.append ( CRLF );
}

response.append ( "Connection: close" );
response.append ( CRLF );

response.append ( CRLF );

byte[] protocol = response.toString().getBytes ( ENCODING_UTF8 );

if ( m_trace != null )
{

```

```

File file = m_trace.createTraceFile ( "HTTP_PROTOCOL_RESPON

JSMHTTPHelper.outputToFile ( file, protocol );
}

/*
Send response

If the client socket is closed, then a broken pipe exception will occur
*/

OutputStream outputStream = m_transport.getOutputStream ();

outputStream.write ( protocol );

/*
Send file content
*/

if ( uncompressContent )
{
if ( m_trace != null )
{
m_trace.println ( "Uncompress content" );
}

inputStream = new GZIPInputStream ( new FileInputStream ( sendF
}
else
{
inputStream = new FileInputStream ( sendFile );
}

if ( sendChunked )
{
JSMHTTPHelper.sendChunked ( inputStream, outputStream );
}
else
{

```

```

        JSMHTTPHelper.sendStream ( inputStream, outputStream ) ;
    }

    inputStream.close () ;

    outputStream.flush () ;
}
catch ( IOException e )
{
    /*
        The user can close the browser, before all the content is sent
    */

    if ( inputStream != null )
    {
        try
        {
            inputStream.close () ;
        }
        catch ( Exception e2 )
        {
        }
    }

    if ( m_trace != null )
    {
        m_trace.println ( "Error sending file response" ) ;
    }

    m_virtual.logError ( m_transport, m_request, "Error sending file respon
}
}

private final boolean isAllowedMethod ( String method )
{
    /*
        Standard HTTP methods

        GET

```

```
    PUT
    POST
    HEAD
    TRACE
    DELETE
    OPTIONS
    CONNECT
*/

if ( method.equals ( "GET" ) )
{
    return true ;
}

if ( method.equals ( "HEAD" ) )
{
    return true ;
}

return false ;
}

private final boolean canAcceptCompressed ()
{
    if ( m_request.canAcceptGZIP () )
    {
        return true ;
    }

    return false ;
}

private final int getCacheAge ( String contentType, File sendFile )
{
    int cacheAge = getCacheAge () ;

    if ( JSMHTTPHelper.isImage ( contentType ) )
    {
        return getCacheAgeImage () ;
    }
}
```

```

    }

    if ( JSMHTTPHelper.isPDF ( contentType ) )
    {
        return getCacheAgePDF () ;
    }

    return cacheAge ;
}

private final int getCacheAge ()
{
    return getServiceParameterInteger ( "CACHE.MAXAGE" ) ;
}

private final int getCacheAgePDF ()
{
    /*
        IE does not pass the pdf content off to the Adobe reader if the cache is (
        Example browser URI /axes/dbmhelp.pdf
    */

    return getServiceParameterInteger ( "CACHE.MAXAGE.PDF" ) ;
}

private final int getCacheAgeImage ()
{
    /*
        YUI/IE image caching

        IE is making frequent requests for images

        Tell the browser to cache the image, the default is no cache
    */

    return getServiceParameterInteger ( "CACHE.MAXAGE.IMAGE" ) ;
}

```

```
private final int getServiceParameterInteger ( String property )
{
    String value = (String)m_serviceParameters.get ( property ) ;

    if ( value == null )
    {
        return 0 ;
    }

    if ( value.equals ( EMPTY_STRING ) )
    {
        return 0 ;
    }

    return Integer.parseInt ( value ) ;
}
}
```

5. Java Service Manager Services

Note: The Service descriptions are presently being revised and extended. Some service descriptions now have an extended description, examples of use, service command syntax diagrams and parameter descriptions. In future versions of this guide, all services will be in this extended format.

The non-extended service descriptions contain a definition of each service including a table listing these items:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

The Command, Keyword and Value are used with the JSMCMD argument in the JSM_COMMAND Built-In Function. The Notes for developers provide some details for the specific keyword or value.

Consider this example from the SMSService table:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	TO	value	Mandatory. Mobile number.
	MSG	value	Mandatory. Text message.

For the example in the above table, this command might appear as follows:

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SEND TO(041123456) M:
```

Some familiarity with the underlying service or protocol may be required in order to properly use the commands. For example, if you are using the SMSService, you may need to SET SMS transport parameters before you can SEND messages.

Services provided are:

5.41 AxesTerminalService	5.11 HTTPOutboundJSONBindService	5.1
5.39 BASE64FileService	5.4 HTTPService	5.2
5.36 ExcelService	5.28 JMSFileService	SC
5.37 ExcelReadService	5.29 JMSXMLBindService	5.2
5.26 FileQueueService	5.30 JSONBindFileService	SC
5.2 FTPService	5.27 OpenLDAPService	5.3
5.42 HashService	5.34 PDFDocumentService	5.3
5.5	5.33 PDFSpoolFileService	5.1
HTTPInboundJSONBindService	5.40 PGPFFileService	XM
5.6 HTTPInboundQueryService	5.14 POP3MailService	5.2
5.9 HTTPInboundSVService	5.31 RFIDDataSourceService	XM
5.8	5.3 SFTPSERVICE	5.1
HTTPInboundXMLBindService	5.15 SMSService	5.1
5.7 HTTPInboundXMLService	5.13	5.2
5.10	SMTPEmailAttachmentSignatureService	XM
HTTPOutboundXMLBindService		5.1
		XM
		5.2
		5.2
		5.3

5.1 Supplementary Information

- 5.1.1 Mapping Service Name to Java Classes
- 5.1.2 IANA Encodings
- 5.1.3 Time Zones
- 5.1.4 Date and Time Formats
- 5.1.5 Decimal Formats
- 5.1.6 Web Browser Content
- 5.1.7 Carriage Return, Line Feed and New Line
- 5.1.8 XML Validation
- 5.1.9 XML Entity Resolver
- 5.1.10 XML Namespace
- 5.1.11 XML Transformation
- 5.1.12 XSL Extension
- 5.1.13 MQSeries Built-In Functions
- 5.1.14 MQSeries IBM i Configuration
- 5.1.15 MQSeries Programs
- 5.1.16 Data Queue Programs
- 5.1.17 IBM Toolbox for Java
- 5.1.18 Common Command Keywords

5.1.1 Mapping Service Name to Java Classes

The service name is the value used by the SERVICE keyword. This name is used in a keyword lookup on the service.properties file in the system subdirectory. The string "service." is prefixed to the service name and the Java class that will supply the service is located and loaded. This loaded class then receives the future commands, until it is unloaded and a new service is loaded.

Example

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVICE')
```

The service FTPService is used by the JSM server to locate the Java class com.lansa.jsm.service.FTPService.

Example

```
service.FTPService=com.lansa.jsm.service.FTPService  
service.XMLParserService=com.lansa.jsm.service.XMLParserService  
service.POP3MailService=com.lansa.jsm.service.POP3MailService
```

5.1.2 IANA Encodings

The HTTP protocol content-type charset encoding name and the XML declaration encoding name use the IANA character set names.

Integrator uses the following tables to correctly encode content and to modify the XML declaration encoding to match the selected byte encoding.

For more information, see IANA encoding and Java Supported encoding:

www.iana.org/assignments/character-sets

<http://download.oracle.com/javase/1.5.0/docs/guide/intl/encoding.doc.html>

The following table contains the Java encoding to IANA encoding mappings.

Java Encoding	IANA encoding
UTF8	utf-8
UTF-8	utf-8
UTF-16BE	utf-16be
UTF-16LE	utf-16le
ASCII	ascii
ISO-8859-1	iso-8859-1
ISO-8859-2	iso-8859-2
ISO-8859-3	iso-8859-3
ISO-8859-4	iso-8859-4
ISO-8859-5	iso-8859-5
ISO-8859-6	iso-8859-6
ISO-8859-7	iso-8859-7
ISO-8859-8	iso-8859-8
ISO-8859-9	iso-8859-9
ISO-8859-13	iso-8859-13
ISO-8859-15	iso-8859-15

ISO8859_1	iso-8859-1
ISO8859_2	iso-8859-2
ISO8859_3	iso-8859-3
ISO8859_4	iso-8859-4
ISO8859_5	iso-8859-5
ISO8859_6	iso-8859-6
ISO8859_7	iso-8859-7
ISO8859_8	iso-8859-8
ISO8859_9	iso-8859-9
ISO8859_13	iso-8859-13
ISO8859_15	iso-8859-15
CP1250	windows-1250
CP1251	windows-1251
CP1252	windows-1252
CP1253	windows-1253
CP1254	windows-1254
CP1255	windows-1255
CP1256	windows-1256
CP1257	windows-1257
MS874	windows-874
MS932	windows-932
MS936	windows-936
MS949	windows-949
MS950	windows-950
CP037	ebcdic-cp-us

CP277	ebcdic-cp-dk
CP278	ebcdic-cp-fi
CP280	ebcdic-cp-it
CP284	ebcdic-cp-es
CP285	ebcdic-cp-gb
CP290	ebcdic-jp-kana
CP297	ebcdic-cp-fr
CP424	ebcdic-cp-he
CP500	ebcdic-cp-ch
CP870	ebcdic-cp-yu
CP871	ebcdic-cp-is
CP918	ebcdic-cp-ar2
CP420	ebcdic-cp-ar1
CP140	ebcdic-us-37+euro
CP141	ebcdic-de-273+euro
CP142	ebcdic-dk-277+euro
CP143	ebcdic-fi-278+euro
CP144	ebcdic-it-280+euro
CP145	ebcdic-es-284+euro
CP146	ebcdic-gb-285+euro
CP147	ebcdic-fr-297+euro
CP148	ebcdic-international-500+euro
CP149	ebcdic-is-871+euro
EUCJIS	euc-jis
EUC_JP	euc-jp

JIS	iso2022-jp
SJIS	Shift_JIS
BIG5	big5
GB2312	gb2312
KOI8-R	koi8-r
KOI8_R	koi8-r
ISO2022KR	euc-kr
CP273	ibm-273
CP437	ibm-437
CP775	ibm-775
CP850	ibm-850
CP852	ibm-852
CP855	ibm-855
CP857	ibm-857
CP860	ibm-860
CP861	ibm-861
CP862	ibm-862
CP863	ibm-863
CP864	ibm-864
CP865	ibm-865
CP866	ibm-866
CP868	ibm-868
CP869	ibm-869
CP1026	ibm-1026
CP1047	ibm-1047

The following table contains the IANA encoding to Java encoding mappings.

IANA Encoding	Java encoding
UTF-8	UTF-8
UTF-16BE	UTF-16BE
UTF-16LE	UTF-16LE
ASCII	ISO8859_1
US-ASCII	ISO8859_1
ISO-8859-1	ISO8859_1
ISO-8859-2	ISO8859_2
ISO-8859-3	ISO8859_3
ISO-8859-4	ISO8859_4
ISO-8859-5	ISO8859_5
ISO-8859-6	ISO8859_6
ISO-8859-7	ISO8859_7
ISO-8859-8	ISO8859_8
ISO-8859-9	ISO8859_9
ISO-8859-13	ISO8859_13
ISO-8859-15	ISO8859_15
ISO_8859_1	ISO8859_1
ISO_8859_2	ISO8859_2
ISO_8859_3	ISO8859_3
ISO_8859_4	ISO8859_4
ISO_8859_5	ISO8859_5

ISO_8859_6	ISO8859_6
ISO_8859_7	ISO8859_7
ISO_8859_8	ISO8859_8
ISO_8859_9	ISO8859_9
ISO_8859_13	ISO8859_13
ISO_8859_15	ISO8859_15
WINDOWS-1250	Cp1250
WINDOWS-1251	Cp1251
WINDOWS-1252	Cp1252
WINDOWS-1253	Cp1253
WINDOWS-1254	Cp1254
WINDOWS-1255	Cp1255
WINDOWS-1256	Cp1256
WINDOWS-1257	Cp1257
WINDOWS-874	MS874
WINDOWS-932	MS932
WINDOWS-936	MS936
WINDOWS-949	MS949
WINDOWS-950	MS950
EBCDIC-CP-US	Cp037
EBCDIC-CP-CA	Cp037
EBCDIC-CP-NL	Cp037
EBCDIC-CP-DK	Cp277
EBCDIC-CP-NO	Cp277
EBCDIC-CP-FI	Cp278

EBCDIC-CP-SE	Cp278
EBCDIC-CP-IT	Cp280
EBCDIC-CP-ES	Cp284
EBCDIC-CP-GB	Cp285
EBCDIC-JP-KANA	Cp290
EBCDIC-CP-FR	Cp297
EBCDIC-CP-HE	Cp424
EBCDIC-CP-CH	Cp500
EBCDIC-CP-YU	Cp870
EBCDIC-CP-IS	Cp871
EBCDIC-CP-AR2	Cp918
EBCDIC-CP-AR1	Cp420
EBCDIC-US-37+EURO	Cp1140
EBCDIC-DE-273+EURO	Cp1141
EBCDIC-DK-277+EURO	Cp1142
EBCDIC-NO-277+EURO	Cp1142
EBCDIC-FI-278+EURO	Cp1143
EBCDIC-SE-278+EURO	Cp1143
EBCDIC-IT-280+EURO	Cp1144
EBCDIC-ES-284+EURO	Cp1145
EBCDIC-GB-285+EURO	Cp1146
EBCDIC-FR-297+EURO	Cp1147
EBCDIC-INTERNATIONAL-500+EURO	Cp1148
EBCDIC-IS-871+EURO	Cp1149
EUCJP	EUC_JP

EUC-JP	EUC_JP
ISO2022-JP	ISO2022JP
ISO-2022-JP	ISO2022JP
SHIFT_JIS	SJIS
SHIFT-JIS	SJIS
IBM932	MS932
IBM-932	MS932
X-SJIS	MS932
WINDOWS-31J	MS932
BIG5	Big5
GB2312	GB2312
KOI8-R	KOI8_R
EUC-KR	EUC_KR
ISO-2022-KR	ISO2022KR
CP367	ISO8859_1
IBM367	ISO8859_1
IBM-367	ISO8859_1
CP819	ISO8859_1
IBM819	ISO8859_1
IBM-819	ISO8859_1
CP1047	ISO8859_1
IBM1047	ISO8859_1
IBM-1047	ISO8859_1
IBM-37	Cp037
IBM-273	Cp273

IBM-277	Cp277
IBM-278	Cp278
IBM-280	Cp280
IBM-284	Cp284
IBM-285	Cp285
IBM-290	Cp290
IBM-297	Cp297
IBM-420	Cp420
IBM-424	Cp424
IBM-437	Cp437
IBM-500	Cp500
IBM-775	Cp775
IBM-850	Cp850
IBM-852	Cp852
IBM-855	Cp855
IBM-857	Cp857
IBM-858	Cp858
IBM-860	Cp860
IBM-861	Cp861
IBM-862	Cp862
IBM-863	Cp863
IBM-864	Cp864
IBM-865	Cp865
IBM-866	Cp866
IBM-868	Cp868

IBM-869	Cp869
IBM-870	Cp870
IBM-871	Cp871
IBM-918	Cp918
IBM-924	Cp924
IBM-1026	Cp1026
IBM-1140	Cp1140
IBM-1141	Cp1141
IBM-1142	Cp1142
IBM-1143	Cp1143
IBM-1144	Cp1144
IBM-1145	Cp1145
IBM-1146	Cp1146
IBM-1147	Cp1147
IBM-1148	Cp1148
IBM-1149	Cp1149

5.1.3 Time Zones

JSM services that use date and time values also have an associated time zone.

The default time zone is supplied by the JVM. The Java System property `user.timezone` can be used to change the JVM's default time zone.

A time zone represents a time offset and daylight savings schedule.

A custom time zone has a time offset and no daylight savings schedule.

The syntax of a custom time zone is:

GMT Sign TwoDigitHours : TwoDigitMinutes

Sign: + or -

Hours: 00 to 23

Minutes: 00 to 59

If you specify an incorrect custom Time Zone ID, then "GMT" is used.

Example

GMT-08:00

GMT+10:30

Australia/Sydney

Time Zone IDs

Africa/Abidjan	America/Regina	Brazil/DeNoronh.
Africa/Accra	America/Rio_Branco	Brazil/East
Africa/Addis_Ababa	America/Rosario	Brazil/West
Africa/Algiers	America/Santiago	Canada/Atlantic
Africa/Asmera	America/Santo_Domingo	Canada/Central
Africa/Bamako	America/Sao_Paulo	Canada/Eastern
Africa/Bangui	America/Scoresbysund	Canada/East-Saskatchewan
Africa/Banjul	America/Shiprock	Canada/Mountair

Africa/Bissau	America/St_Johns	Canada/Newfoun
Africa/Blantyre	America/St_Kitts	Canada/Pacific
Africa/Brazzaville	America/St_Lucia	Canada/Saskatche
Africa/Bujumbura	America/St_Thomas	Canada/Yukon
Africa/Cairo	America/St_Vincent	Chile/Continental
Africa/Casablanca	America/Swift_Current	Chile/EasterIslan
Africa/Ceuta	America/Tegucigalpa	Cuba
Africa/Conakry	America/Thule	Egypt
Africa/Dakar	America/Thunder_Bay	Eire
Africa/Dar_es_Salaam	America/Tijuana	Europe/Amsterda
Africa/Djibouti	America/Tortola	Europe/Andorra
Africa/Douala	America/Vancouver	Europe/Athens
Africa/El_Aaiun	America/Virgin	Europe/Belfast
Africa/Freetown	America/Whitehorse	Europe/Belgrade
Africa/Gaborone	America/Winnipeg	Europe/Berlin
Africa/Harare	America/Yakutat	Europe/Bratislava
Africa/Johannesburg	America/Yellowknife	Europe/Brussels
Africa/Kampala	Antarctica/Casey	Europe/Bucharest
Africa/Khartoum	Antarctica/Davis	Europe/Budapest
Africa/Kigali	Antarctica/DumontDUrville	Europe/Chisinau
Africa/Kinshasa	Antarctica/Mawson	Europe/Copenhag
Africa/Lagos	Antarctica/McMurdo	Europe/Dublin
Africa/Libreville	Antarctica/Palmer	Europe/Gibraltar
Africa/Lome	Antarctica/Rothera	Europe/Helsinki

Africa/Luanda	Antarctica/South_Pole	Europe/Istanbul
Africa/Lubumbashi	Antarctica/Syowa	Europe/Kaliningr
Africa/Lusaka	Antarctica/Vostok	Europe/Kiev
Africa/Malabo	Arctic/Longyearbyen	Europe/Lisbon
Africa/Maputo	Asia/Aden	Europe/Ljubljana
Africa/Maseru	Asia/Almaty	Europe/London
Africa/Mbabane	Asia/Amman	Europe/Luxembo
Africa/Mogadishu	Asia/Anadyr	Europe/Madrid
Africa/Monrovia	Asia/Aqtou	Europe/Malta
Africa/Nairobi	Asia/Aqtobe	Europe/Minsk
Africa/Ndjamena	Asia/Ashgabat	Europe/Monaco
Africa/Niamey	Asia/Ashkhabad	Europe/Moscow
Africa/Nouakchott	Asia/Baghdad	Europe/Nicosia
Africa/Ouagadougou	Asia/Bahrain	Europe/Oslo
Africa/Porto-Novo	Asia/Baku	Europe/Paris
Africa/Sao_Tome	Asia/Bangkok	Europe/Prague
Africa/Timbuktu	Asia/Beirut	Europe/Riga
Africa/Tripoli	Asia/Bishkek	Europe/Rome
Africa/Tunis	Asia/Brunei	Europe/Samara
Africa/Windhoek	Asia/Calcutta	Europe/San_Mari
America/Adak	Asia/Choibalsan	Europe/Sarajevo
America/Anchorage	Asia/Chongqing	Europe/Simferopol
America/Anguilla	Asia/Chungking	Europe/Skopje
America/Antigua	Asia/Colombo	Europe/Sofia

America/Araguaina	Asia/Dacca	Europe/Stockholm
America/Aruba	Asia/Damascus	Europe/Tallinn
America/Asuncion	Asia/Dhaka	Europe/Tirane
America/Atka	Asia/Dili	Europe/Tiraspol
America/Barbados	Asia/Dubai	Europe/Uzhgorod
America/Belem	Asia/Dushanbe	Europe/Vaduz
America/Belize	Asia/Gaza	Europe/Vatican
America/Boa_Vista	Asia/Harbin	Europe/Vienna
America/Bogota	Asia/Hong_Kong	Europe/Vilnius
America/Boise	Asia/Hovd	Europe/Warsaw
America/Buenos_Aires	Asia/Irkutsk	Europe/Zagreb
America/Cambridge_Bay	Asia/Istanbul	Europe/Zaporozh
America/Cancun	Asia/Jakarta	Europe/Zurich
America/Caracas	Asia/Jayapura	Greenwich
America/Catamarca	Asia/Jerusalem	Hongkong
America/Cayenne	Asia/Kabul	Iceland
America/Cayman	Asia/Kamchatka	Indian/Antananar
America/Chicago	Asia/Karachi	Indian/Chagos
America/Chihuahua	Asia/Kashgar	Indian/Christmas
America/Cordoba	Asia/Katmandu	Indian/Cocos
America/Costa_Rica	Asia/Krasnoyarsk	Indian/Comoro
America/Cuiaba	Asia/Kuala_Lumpur	Indian/Kerguelen
America/Curacao	Asia/Kuching	Indian/Mahe
America/Danmarkshavn	Asia/Kuwait	Indian/Maldives

America/Dawson	Asia/Macao	Indian/Mauritius
America/Dawson_Creek	Asia/Macau	Indian/Mayotte
America/Denver	Asia/Magadan	Indian/Reunion
America/Detroit	Asia/Makassar	Iran
America/Dominica	Asia/Manila	Israel
America/Edmonton	Asia/Muscat	Jamaica
America/Eirunepe	Asia/Nicosia	Japan
America/El_Salvador	Asia/Novosibirsk	Kwajalein
America/Ensenada	Asia/Omsk	Libya
America/Fort_Wayne	Asia/Oral	Mexico/BajaNort
America/Fortaleza	Asia/Phnom_Penh	Mexico/BajaSur
America/Glace_Bay	Asia/Pontianak	Mexico/General
America/Godthab	Asia/Pyongyang	Navajo
America/Goose_Bay	Asia/Qatar	Pacific/Apia
America/Grand_Turk	Asia/Qyzylorda	Pacific/Auckland
America/Grenada	Asia/Rangoon	Pacific/Chatham
America/Guadeloupe	Asia/Riyadh	Pacific/Easter
America/Guatemala	Asia/Saigon	Pacific/Efate
America/Guayaquil	Asia/Sakhalin	Pacific/Enderbury
America/Guyana	Asia/Samarkand	Pacific/Fakaofu
America/Halifax	Asia/Seoul	Pacific/Fiji
America/Havana	Asia/Shanghai	Pacific/Funafuti
America/Hermosillo	Asia/Singapore	Pacific/Galapagos
America/Indiana/Indianapolis	Asia/Taipei	Pacific/Gambier

America/Indiana/Knox	Asia/Tashkent	Pacific/Guadalca
America/Indiana/Marengo	Asia/Tbilisi	Pacific/Guam
America/Indiana/Vevay	Asia/Tehran	Pacific/Honolulu
America/Indianapolis	Asia/Tel_Aviv	Pacific/Johnston
America/Inuvik	Asia/Thimbu	Pacific/Kiritimati
America/Iqaluit	Asia/Thimphu	Pacific/Kosrae
America/Jamaica	Asia/Tokyo	Pacific/Kwajaleir
America/Jujuy	Asia/Ujung_Pandang	Pacific/Majuro
America/Juneau	Asia/Ulaanbaatar	Pacific/Marquesa
America/Kentucky/Louisville	Asia/Ulan_Bator	Pacific/Midway
America/Kentucky/Monticello	Asia/Urumqi	Pacific/Nauru
America/Knox_IN	Asia/Vientiane	Pacific/Niue
America/La_Paz	Asia/Vladivostok	Pacific/Norfolk
America/Lima	Asia/Yakutsk	Pacific/Noumea
America/Los_Angeles	Asia/Yekaterinburg	Pacific/Pago_Pag
America/Louisville	Asia/Yerevan	Pacific/Palau
America/Maceio	Atlantic/Azores	Pacific/Pitcairn
America/Managua	Atlantic/Bermuda	Pacific/Ponape
America/Manaus	Atlantic/Canary	Pacific/Port_Mor
America/Martinique	Atlantic/Cape_Verde	Pacific/Rarotonga
America/Mazatlan	Atlantic/Faeroe	Pacific/Saipan
America/Mendoza	Atlantic/Jan_Mayen	Pacific/Samoa
America/Menominee	Atlantic/Madeira	Pacific/Tahiti
America/Merida	Atlantic/Reykjavik	Pacific/Tarawa

America/Mexico_City	Atlantic/South_Georgia	Pacific/Tongatapu
America/Miquelon	Atlantic/St_Helena	Pacific/Truk
America/Monterrey	Atlantic/Stanley	Pacific/Wake
America/Montevideo	Australia/ACT	Pacific/Wallis
America/Montreal	Australia/Adelaide	Pacific/Yap
America/Montserrat	Australia/Brisbane	Poland
America/Nassau	Australia/Broken_Hill	Portugal
America/New_York	Australia/Canberra	Singapore
America/Nipigon	Australia/Darwin	Turkey
America/Nome	Australia/Hobart	US/Alaska
America/Noronha	Australia/Lindeman	US/Aleutian
America/North_Dakota/Center	Australia/Lord_Howe	US/Arizona
America/Panama	Australia/Melbourne	US/Central
America/Pangnirtung	Australia/North	US/Eastern
America/Paramaribo	Australia/NSW	US/East-Indiana
America/Phoenix	Australia/Perth	US/Hawaii
America/Port_of_Spain	Australia/Queensland	US/Indiana-Stark
America/Port-au-Prince	Australia/South	US/Michigan
America/Porto_Acre	Australia/Sydney	US/Mountain
America/Porto_Velho	Australia/Tasmania	US/Pacific
America/Puerto_Rico	Australia/Victoria	US/Pacific-New
America/Rainy_River	Australia/West	US/Samoa
America/Rankin_Inlet	Australia/Yancowinna	Zulu
America/Recife	Brazil/Acre	

5.1.4 Date and Time Formats

The International Standard for the representation of dates and times is ISO8601. For unambiguous representation of dates and times the following formats have been chosen. The date separator - (dash) can be replaced with a / (slash). The time element separator T can be a blank. The time zone value of hh:mm can be hhmm or hh, but the + or - is mandatory. If no time zone is present then a hierarchy of time zones will be used, the time zone determined by the service or +00:00.

A program can pass a date value to the service using different formats. If the program field is numeric then use the all numbers format. If program field is an alphanumeric then use the string format or numeric format.

Possible program field values:

YYYYMMDD [TZD]

YYYYMMDDHHMMSS [TZD]

YYYY-MM-DD [TZD]

YYYY/MM/DD [TZD]

YYYY-MM-DD HH:MM:SS [TZD]

YYYY-MM-DDTHH:MM:SS [TZD]

YYYY-MM-DDTHH:MM:SS.SSSSSSSSS [TZD]

YYYY Four digit year (1000-9999).

MM Two digit month (01-12).

DD Two digit day (01-31).

HH Two digit hour (00-23).
Default. 00.

MM Two digit minute (00-59).
Default 00.

SS Two digit second (00-59).
Default 00.

SSSSSSSSSS One to nine digits representing a decimal fraction of a second.
Default. 000000000.

TZD Time zone designator (Z or +hh:mm or -hh:mm)
 Z means GMT or +00:00.
 Default. +00:00.

Example program field date values:

20040331
 20040331+10
 20040331150000
 20040331150000Z
 20040331150000+10
 20040331150000+1000
 20040331150000+10:00

2004-03-31
 2004-03-31+10
 2004-03-31 15:00:00
 2004-03-31 15:00:00Z
 2004-03-31 15:00:00+10
 2004-03-31 15:00:00+1000
 2004-03-31 15:00:00+10:00
 2004-03-31 15:00:00.000+10

2004-03-31T15:00:00.000+10:00

To convert a date time into a string representation the following case-sensitive date format patterns can be used.

yy	2 digit year	00-99.
yyyy	4 digit year	1000-9999.
MM	Month in year	01-12.
MMM	Month in year	Jan-Dec (Locale dependent).
MMMM	Month in year	January-December (Locale dependent).
dd	Day in month	01-31.

HH	Hour in day	00-23.
mm	Minute in hour	00-59.
ss	Second in minute	00-59.
SSS	Millisecond	000.
SSSSSSSSS	Nanosecond	000000000.
z	TimeZone	GMT+00:00.
Z	TimeZone	+0000.
ZZ	TimeZone	+00:00.
AM/PM	AM PM Marker	AM or PM (Locale dependant).
hh	Hour in AM/PM	1-12.
EE	Day in week	1-7.
EEE	Day in week	Sun-Sat (Locale dependant).
EEEE	Day in week	Sunday-Saturday (Locale dependant).

Example date format patterns:

dd-MM-yyyy

dd-MM-yyyy HH:mm:ss

dd-MM-yyyy HH:mm:ss Z

5.1.5 Decimal Formats

Decimal format patterns allow numbers to be formatted to a particular locale.

- 0 Digit.
- # Digit, zero shows as absent.
- . Decimal separator or monetary decimal separator.
- Minus sign.
- , Grouping separator.
- E Separates mantissa and exponent in scientific notation.
- ; Subpattern boundary.
- % Multiply by 100 and show as percentage.
- \u2030 Multiply by 1000 and show as per mile.
- \u00A4 Currency sign, replaced by currency symbol.
If doubled, replaced by international currency symbol.
If present in a pattern, the monetary decimal separator is used instead of the decimal separator.
If used in an XML attribute, use hex entity `¤`;
- ' Used to quote special characters.
If the % character is needed in the pattern then use '%'.
If the # character is needed in the pattern then use '#'
To create a single quote itself, use two in a row.
The single quote can be used inside a double quoted XML attribute.

5.1.6 Web Browser Content

Two content encoding methods are available when sending data from a HTTP client browser to a HTTP server.

Both methods do not identify the byte encoding used by the client browser so the conversion of client data to the encoding of the server can be problematic if the characters are not in the ASCII range.

It has been observed that some browsers will use the content-type charset attribute to override the default client browser encoding that is applied to the INPUT form data being posted to the server.

If the HTML page has been served from the server using a content-type charset attribute then this will have the same effect as a meta tag within the HTML document.

To control the byte encoding used by the HTTP client browser using a meta tag in the HTML document.

```
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
</head>
```

Using a charset value of utf-8 means that the HTML FORM post from any client locale will be received by the server UTF-8 encoded.

Method 1: multipart/format-data

HTML source:

```
<FORM METHOD="POST" ACTION="http://lansa01:88/cgi-bin/jsmdirect?
upload" ENCTYPE="multipart/form-data">
<INPUT NAME="SONUMBER" TYPE="TEXT" VALUE="12345"/>
</TD></TR>
<INPUT NAME="CUSTNAME" TYPE="TEXT" VALUE="ABC Industries"
</TD></TR>
<INPUT NAME="STREET" TYPE="TEXT" VALUE="123 Main St"/>
</TD></TR>
<INPUT NAME="CITY" TYPE="TEXT" VALUE="Chicago"/></TD>
</TR>
<INPUT NAME="POSTCODE" TYPE="TEXT" VALUE="60609"/></TD>
```

```
</TR>
<INPUT NAME="FILE" TYPE="FILE" SIZE="60"/>
<INPUT TYPE="SUBMIT" VALUE="Send"/>
</FORM>
```

Content type:

multipart/form-data; boundary=-----7d37e321500b2

Content:

-----7d37e321500b2
Content-Disposition: form-data; name="SONUMBER"

12345

-----7d37e321500b2
Content-Disposition: form-data; name="CUSTNAME"

ABC Industries

-----7d37e321500b2
Content-Disposition: form-data; name="STREET"

123 Main St

-----7d37e321500b2
Content-Disposition: form-data; name="CITY"

Chicago

-----7d37e321500b2
Content-Disposition: form-data; name="POSTCODE"

60609

-----7d37e321500b2
Content-Disposition: form-data; name="FILE"; filename=""
Content-Type: application/octet-stream

-----7d37e321500b2--

Method 2: application/x-www-form-urlencoded

HTML source:

```
<FORM METHOD="POST" ACTION="http://lansa01:88/cgi-bin/jsmdirect?
upload">
<INPUT NAME="ORDER" TYPE="TEXT" VALUE="12345"/></TD>
</TR>
<INPUT NAME="CUSTNAME" TYPE="TEXT" VALUE="ABC Industries"
</TD></TR>
<INPUT NAME="STREET" TYPE="TEXT" VALUE="123 Main St"/>
</TD></TR>
<INPUT NAME="CITY" TYPE="TEXT" VALUE="Chicago"/></TD>
</TR>
<INPUT NAME="POSTCODE" TYPE="TEXT" VALUE="60609"/></TD>
</TR>
<INPUT TYPE="SUBMIT" VALUE="Send"/>
</FORM>
```

Content type:

application/x-www-form-urlencoded

Content:

ORDER=12345&CUSTNAME=ABC+Industries&STREET=123+Main+St&C

5.1.7 Carriage Return, Line Feed and New Line

New lines are represented on different platforms by carriage return (CR), line feed (LF), CRLF, or new line (NEL).

Unfortunately, not only are new lines represented by different characters on different platforms, they also have ambiguous behaviour even on the same platform.

Especially with the advent of the web, where text on a single machine can arise from many sources, this causes a significant problem.

Unfortunately, these characters are often transcoded directly into the corresponding Unicode codes when a character set is transcoded; this means that even programs handling pure Unicode have to deal with the problems.

	Unicode	ASCII	EBCDIC 1	EBCDIC 2
CR	000D	0D	0D	0D
LF	000A	0A	25	15
CRLF	000D 000A	0D 0A	0D 25	0D 15
NEL	0085	85	15	25
VT	000B	0B	0B	0B
FF	000C	0C	0C	0C
LS	2028	n/a	n/a	n/a
PS	2029	n/a	n/a	n/a

There are two mappings of LF and NEL used by EBCDIC systems.

The first EBCDIC column shows the MVS Open Edition (including CP1047) mapping of these characters, while the second column shows the CDRA mapping.

This difference arises from the use of LF character as 'New Line' in ASCII-based Linux environments and in some data transfer protocols that use the Linux assumptions.

The second column is based on the standardized definitions — both in ASCII and EBCDIC of LF.

NEL is not actually defined in ASCII: it is defined in ISO 6429 as a C1 control.

For more information refer to:

www.w3.org/TR/newline

www.unicode.org/unicode/reports/tr13/tr13-5.html

5.1.8 XML Validation

All XML services have the ability to validate XML documents using DTD and XML Schema grammars.

All XML services use the SAX2 'org.xml.sax.XMLReader' interface and validation features are enabled via this interface.

All XML services use the 'org.apache.xerces.parsers.SAXParser' parser and have the following features enabled.

Feature	Description
http://xml.org/sax/features/validation	DTD validation is enabled.
http://xml.org/sax/features/namespace	Name space processing is enabled.
http://apache.org/xml/features/validation/dynamic	Dynamic validation is enabled. Validation is only done if the grammar is present in the document.
http://apache.org/xml/features/validation/schema	XML schema validation is disabled by default. This feature can be enabled by the use of the SCHEMA keyword or 'validation.schema' property.

The DTD grammar is specified by the DOCTYPE before the root element.

```
<!DOCTYPE Orders SYSTEM "order.dtd">
```

The XML schema grammar is specified by the xsi:schemaLocation or xsi:noNamespaceSchemaLocation attribute on the root element of the document.

The xsi prefix must be bound to the Schema document instance namespace, as specified by the recommendation.

Each document that uses XML Schema grammars must specify the location of the grammars it uses by using an xsi:schemaLocation attribute if they use namespaces, and an xsi:noNamespaceSchemaLocation attribute otherwise.

These are usually placed on the root / top-level element in the document, though they may occur on any element.

Here is an example with no target namespace:

```
<document xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='document.xsd'>
</document>
```

Here is an example with a target namespace. Note that it is an error to specify a different namespace than the target namespace defined in the Schema.

```
<document xmlns='http://my.com' xmlns:xsi='http://www.w3.org/2001/XMLSchema
instance'
  xsi:schemaLocation='http://my.com document.xsd'>
</document>
```

What happens if both DTD validation and schema validation features are on ?

If both validators are present in the pipeline, then

If the instance document has only a DTD grammar (DOCTYPE before the root element), then only DTD validation errors are reported.

If the instance document has only XML Schema grammars, then only XML Schema validation errors are reported.

If the instance document has both DTD and XML Schema grammars, validation errors for both DTD and XML Schema are reported.

For more information refer to:

xml.apache.org/xerces2-j/faqs.html

xml.apache.org/xerces2-j/faq-pcftp.html

xml.apache.org/xerces2-j/features.html

xml.apache.org/xerces2-j/xml-schema.html

5.1.9 XML Entity Resolver

All XML services have an entity resolver that allows the parser to locate resources that are external to the document.

The entity resolver can be used to cache external resource files on the local machine, so the service does not need to use the HTTP protocol to get remote resource files. This technique can also be used to handle proxy/firewall issues where the service cannot connect to the remote server.

Refer to xml.apache.org/commons/components/resolver/resolver-article.html

A resource lookup hierarchy is used to determine the location of dtd and schema files.

If a `systemid.{url}` property exists then it will be used.

The following steps are done for file (non-http) resources.

If the 'schema.dir' property exists, then this directory value and the url resource filename are used to determine the location of the file.

If the 'schema.dir' property does not exist and the url resource path does exist, then the url resource path is used.

If the 'schema.dir' property does not exist and the url resource path does not exist, then the instance dtd sub-directory and the url resource filename is used.

Here is an example of an XML document that will use the entity resolver to locate the external resource 'order.dtd'.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Orders SYSTEM "order.dtd">
<Orders>
</Orders>
```

Sample XML service properties entries

`schema.dir=schema`

`systemid.file:///jsm/instance/order.dtd=dtd/order.dtd`

`systemid.file:///jsm/instance/shipment.xsd=dtd/shipment.xsd`

`systemid.http://nasdaq.com/reference/NasdaqDotCom.dtd=dtd/nasdaq.dtd`

`systemid.file:///C:/Program%20Files/LANSA/Integrator/JSMInstance/order.dtd`

Note: The %20 represents a blank.

5.1.10 XML Namespace

XML namespaces provide a simple method for qualifying element and attribute names used in Extensible Markup Language documents by associating them with namespaces identified by URI references.

Refer to [Namespaces in XML 1.0](#)

Apache recommend avoiding the use of default namespaces.

If you are looking for nodes in a namespace, the XPath expression must include a namespace prefix that you have mapped to the namespace with an xmlns declaration.

If you have declared a default namespace, it does not have a prefix.

In order to construct XPath expressions to retrieve nodes from this namespace, you must add a namespace declaration that provides a prefix you can include in the XPath expressions.

Suppose, for example, you want to locate nodes in a default namespace declared as follows:

```
xmlns="http://my-namespace"
```

Add a namespace declaration with a prefix to the style sheet `xsl:transform` element:

```
xmlns:xyz="http://my-namespace"
```

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:rdml="http://www.lansa.com/2000/XML/Function"
  xmlns:xalan="http://xml.apache.org/xalan"
  xmlns:xyz="http://my-namespace">
```

Then you can use `xyz:` in your XPath expression:

```
<rdml:field name="FIELD" value="{/xyz:order/@name}"/>
```

For more information refer to:

[Apache frequently asked questions](#)

5.1.11 XML Transformation

All XML services have been designed so a developer can create transformation style sheets to bind data into and out of the XML document.

The [XML Transformation Wizard](#) creates the required transformation style sheets to perform this task.

It is possible to use these created XSLT files as a foundation and add additional XSLT instructions. This task needs to be approached with caution as the XSLT syntax is difficult and performance can easily be degraded if incorrect design decisions are made.

For improved transformation performance convert the XSLT files into compiled transformation style sheets using the [XSL Compiler](#).

For more information refer to:

[W3C: XSL Transformations \(XSLT\)](#)

[W3C: XML Path Language \(XPath\)](#)

5.1.12 XSL Extension

For those situations where you would like to augment the functionality of XSLT with calls to a procedural language, Apache Xalan supports the creation and use of extension elements and extension functions.

Refer to xml.apache.org/xalan-j/extensions.html

[Xalan-Java extensions](#)

Compiled transformation style sheets (XSLTC) support the use of extension functions implemented in external Java classes.

The following example illustrates how to create and use an extension function.

XML source:

```
<?xml version="1.0" encoding="UTF-8"?>

<Orders>
  <SalesOrder SONumber="">
    <Customer CustNumber="543">
      <CustName>ABC Industries</CustName>
      <Street>123 Main St</Street>
      <City>Chicago</City>
      <State>IL</State>
      <PostCode>60609</PostCode>
    </Customer>
  </SalesOrder>
</Orders>
```

Transformation style sheet:

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transf
  xmlns:rdml="http://www.lansa.com/2000/XML/Function"
  xmlns:xalan="http://xml.apache.org/xalan"
  xmlns:java="http://xml.apache.org/xalan/java"
  exclude-result-prefixes="xalan java">

<xsl:output method="xml" indent="yes" xalan:indent-amount="2"/>
```

```
<xsl:template match="/">

<rdml:function>

<rdml:fields>
  <xsl:call-template name="function-level"/>
</rdml:fields>

</rdml:function>

</xsl:template>

<xsl:template name="function-level">

<rdml:field name="DELIVERY" value="
{java:com.acme.xsl.Extension.getValue(/Orders/SalesOrder/@Delivery,'MISSI

<rdml:field name="ORDER" value="
{java:com.acme.xsl.Extension.getValue(/Orders/SalesOrder/@SONumber,'MI

<rdml:field name="STATEDESC" value="
{java:com.acme.xsl.Extension.getDescription(/Orders/SalesOrder/Customer/St

<rdml:field name="CUSTNUM" value="
{/Orders/SalesOrder/Customer/@CustNumber}"/>
<rdml:field name="NAME" value="
{/Orders/SalesOrder/Customer/CustName}"/>
<rdml:field name="STREET" value="
{/Orders/SalesOrder/Customer/Street}"/>
<rdml:field name="CITY" value="{/Orders/SalesOrder/Customer/City}"/>
<rdml:field name="STATE" value="{/Orders/SalesOrder/Customer/State}"/>
<rdml:field name="ZIP" value="{/Orders/SalesOrder/Customer/PostCode}"/>

</xsl:template>

</xsl:transform>
```

XML transformation:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdml:function xmlns:rdml="http://www.lansa.com/2000/XML/Function">
  <rdml:fields>
    <rdml:field value="MISSING" name="DELIVERY"/>
    <rdml:field value="BLANK" name="ORDER"/>
    <rdml:field value="Illinois" name="STATEDESC"/>
    <rdml:field value="543" name="CUSTNUM"/>
    <rdml:field value="ABC Industries" name="NAME"/>
    <rdml:field value="123 Main St" name="STREET"/>
    <rdml:field value="Chicago" name="CITY"/>
    <rdml:field value="IL" name="STATE"/>
    <rdml:field value="60609" name="ZIP"/>
  </rdml:fields>
</rdml:function>
```

Java extension:

```
package com.acme.xsl ;

import org.w3c.dom.* ;

public class Extension
{

    public static String getDescription ( String code )
    {
        if ( code.equalsIgnoreCase ( "IL" ) )
        {
            return "Illinois" ;
        }

        return "?" ;
    }

    public static String getValue ( NodeList nodeList, String missingValue )
    {
```

```

    return getValue ( nodeList, missingValue, "" );
}

public static String getValue ( NodeList nodeList, String missingValue, String blankValue )
{
    int length = nodeList.getLength () ;

    if ( length == 0 )
    {
        return missingValue ;
    }

    Node node = nodeList.item ( 0 ) ;

    String nodeValue = node.getNodeValue () ;

    if ( nodeValue == null )
    {
        return blankValue ;
    }

    if ( nodeValue.trim().equals ( "" ) )
    {
        return blankValue ;
    }

    return nodeValue ;
}
}

```

5.1.13 MQSeries Built-In Functions

The MQSeries Built-In Functions allow messages to be read from a message queue and transferred across to the loaded service for processing. Also XML responses can be transferred from the loaded service and put into a message queue.

All Built-in functions return at least a status and message. The status is a string value or an MQ API reason code prefixed with the string MQR.

- OK
- ERROR
- FATAL
- MQR2002 - Already connected
- MQR2033 - No messages
- MQR2058 - Queue manager name not valid or not known
- MQR2086 - Unknown Queue Manager
- MQR2079 - Truncated message accepted
- MQR2080 - Truncated message failed
- etc..

The connection and queue handle parameters are 4 byte binary values and cannot be sent across to the loaded service, so the fields that hold these values must be named with the JSM prefix to exclude them from the service field list data transfer.

The Built-In Functions are:

[MQ_CONN](#)

[MQ_DISC](#)

[MQ_BEGIN](#)

[MQ_CMIT](#)

[MQ_BACK](#)

[MQ_OPEN](#)

[MQ_CLOSE](#)

[MQ_GET](#)

[MQ_PUT](#)

[MQ_DEPTH](#)

Also see

[MQSeries Built-in Function Code Example](#)

MQ_CONN

The MQ_CONN BIF opens a connection to a queue manager and returns a connection handle.

If the returned status is OK or MQR2002 then the connection handle is usable.

If the returned status is OK then there are no other opened connections and the program should close the connection.

If the returned status is MQR2002 then there are other opened connections and the program should not close the connection, as this will close the other connections as well.

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Decim
1	A	Required	Queue Manager.	48	48		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Decim
1	A	Required	Status.	20	20		
2	A	Required	Message.	1	256		
3	A	Required	Connection Handle.	4	4		

MQ_DISC

The MQ_DISC BIF closes the connection to the queue manager.

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Connection Handle.	4	4		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Status.	20	20		
2	A	Required	Message.	1	256		

MQ_BEGIN

The MQ_BEGIN BIF begins a unit of work that is coordinated by the queue manager. The status return value of MQ2121 (MQRC_NO_EXTERNAL_PARTICIPANTS) can be treated as a notice message and ignored.

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Decim
1	A	Required	Connection Handle.	4	4		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Decim
1	A	Required	Status.	20	20		
2	A	Required	Message.	1	256		

MQ_CMtT

The MQ_CMtT BIF indicates to the queue manager that the application has reached a syncpoint, and that all the message gets and puts that have occurred since the last syncpoint are to be made permanent.

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Connection Handle.	4	4		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Status.	20	20		
2	A	Required	Message.	1	256		

MQ_BACK

The MQ_BACK BIF indicates to the queue manager that all message gets and puts that have occurred since the last syncpoint are to be backed out.

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Connection Handle.	4	4		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Status.	20	20		
2	A	Required	Message.	1	256		

MQ_OPEN

The MQ_OPEN BIF opens a message queue and returns a queue handle.

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Connection Handle.	4	4		
2	A	Required	Queue Name.	48	48		
3	A	Required	Open Mode.	10	10		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Status.	20	20		
2	A	Required	Message.	1	256		
3	A	Required	Queue Handle.	4	4		

The queue can be opened in several modes.

Mode	MQSeries API options
*BLANKS	MQOO_INPUT_AS_Q_DEF MQOO_FAIL_IF_QUIESCING
I	MQOO_INPUT_AS_Q_DEF

Q	MQOO_INQUIRE
IQ	MQOO_INPUT_AS_Q_DEF MQOO_INQUIRE
IF	MQOO_INPUT_AS_Q_DEF MQOO_FAIL_IF QUIESCING
IQF	MQOO_INPUT_AS_Q_DEF MQOO_INQUIRE MQOO_FAIL_IF QUIESCING
O	MQOO_OUTPUT
OF	MQOO_OUTPUT MQOO_FAIL_IF QUIESCING
OU	MQOO_OUTPUT MQOO_SET_IDENTITY_CONTEXT
OFU	MQOO_OUTPUT MQOO_FAIL_IF QUIESCING MQOO_SET_IDENTITY_CONTEXT
IO	MQOO_INPUT_AS_Q_DEF MQOO_OUTPUT
IOF	MQOO_INPUT_AS_Q_DEF MQOO_OUTPUT MQOO_FAIL_IF QUIESCING
IOFU	MQOO_INPUT_AS_Q_DEF MQOO_OUTPUT MQOO_FAIL_IF QUIESCING MQOO_SET_IDENTITY_CONTEXT

MQ_CLOSE

The MQ_CLOSE BIF closes the queue specified by the handle.

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Connection Handle.	4	4		
2	A	Required	Queue Handle.	4	4		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Status.	20	20		
2	A	Required	Message.	1	256		

MQ_GET

The MQ_GET BIF gets a message from the queue and passes it to the loaded service.

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Ma Dec
1	A	Required	Connection Handle.	4	4		
2	A	Required	Queue Handle.	4	4		
3	P	Required	Wait Time (milliseconds) Use -1 for forever.	9	9	0	0
4	P	Required	Initial Message Length Default is 20000 bytes. This is a helper size, if the size is too small the internal storage is reallocated to the size of the message and another get operation is performed.	9	9	0	0
5	A	Required	Message Id Use *BLANK for any	24	24		

			message.				
6	A	Required	Correlation Id User *BLANK for any message.	24	24		
7	A	Optional	SyncPoint Y or N Default is N.	1	1		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Status.	20	20		
2	A	Required	Message.	1	256		
3	P	Required	Message Type. 1 - Request 2 - Reply 4 - Report 8 - Datagram 65536- 999999999 - User defined.	9	9	0	0
4	A	Required	Message Format.	8	8		
5	A	Required	Message Id.	24	24		
6	A	Required	Correlation Id.	24	24		
7	A	Required	Remote	48	48		

			Queue Manager.				
8	A	Required	Remote Queue.	48	48		
9	A	Required	User.	12	12		
10	A	Required	Application Id.	32	32		

MQ_PUT

The MQ_PUT BIF gets a message response from the loaded service and puts it into the queue.

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal
1	A	Required	Connection Handle.	4	4	
2	A	Required	Queue Handle.	4	4	
3	P	Required	Message Type. 8 - Datagram 65536-999999999 - User defined. If a value of 0 is passed then a value of 8 is used.	9	9	0
4	A	Required	Message Format. *BLANK - MQFMT_NONE MQSTR - MQFMT_STRING or user defined.	8	8	
5	A	Required	Message Id or *BLANK for queue manager to auto-generate.	24	24	
6	A	Required	Correlation Id or *BLANK.	24	24	
7	A	Required	Remote Message Queue or *BLANK.	48	48	
8	A	Required	Remote Queue or	48	48	

			*BLANK			
9	P	Optional	Expiry. (millisecond) -1 for queue default.	9	9	0
10	A	Optional	User. Default is *BLANK.	12	12	
11	A	Optional	Application Id.	32	32	

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Status.	20	20		
2	A	Required	Message.	1	256		

MQ_DEPTH

The MQ_DEPTH BIF returns the number of messages in the queue. The queue must be opened with an MQOO_INQUIRE option.

Arguments

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Connection Handle.	4	4		
2	A	Required	Queue Handle.	4	4		

Return Values

No	Type	Required Optional	Description	Min Length	Max Length	Min Decimal	Max Deci
1	A	Required	Status.	20	20		
2	A	Required	Message.	1	256		
3	P	Required	Message Depth.	9	9	0	0

5.1.14 MQSeries IBM i Configuration

The following i5/OS commands illustrate how to create the necessary MQ objects to support MQSeries Java clients.

```
GO CMDMQM
STRSBS QMQM/QMQM
CRTMQM MQMNAME(USERAGENT.QUEUE.MANAGER)
STRMQM MQMNAME(USERAGENT.QUEUE.MANAGER)
CRTMQMQ QNAME(USERAGENT.QUEUE) QTYPE(*LCL) MQMNAME(
CRTMQMCHL CHLNAME(USERAGENT.CHANNEL) CHLTYPE(*SVRCI
STRMQMCHL CHLNAME(USERAGENT.CHANNEL) MQMNAME(USER
STRMQMLSR MQMNAME(USERAGENT.QUEUE.MANAGER)
```

5.1.15 MQSeries Programs

All programs will return at least a status and message. The status is a string value or an MQ API reason code prefixed with the string MQR.

- OK
- ERROR
- FATAL
- MQR2058 - Queue manager name not valid or not known
- MQR2086 - Unknown Queue Manager
- etc..

JSMMQDEPTH - Depth of queue

Parameter	Description	I/O	Type	Len	Dec
JSMQMGR	Queue Manager	I	CHAR	48	
JSMQNAME	Queue	I	CHAR	48	
JSMDEPTH	Queue depth	O	DEC	8	0
JSMSTS	Status	O	CHAR	20	
JSMMSG	Message	O	CHAR	256	

The JSMMQDEPTH program is standalone and can be called without a JSM service being opened.

Example

```
CALL PGM(JSMMQDEPTH) PARM(#JSMQMGR #JSMQNAME #JSMDEI
```

JSMMQGET - Get message from queue

Parameter	Description	I/O	Type	Len	Dec
JSMQMGR	Queue Manager	I	CHAR	48	
JSMQNAME	Queue	I	CHAR	48	

JSMWAIT	Wait time (millisecs, -1 for unlimited.)	I	DEC	8	0
JSMMSIZE	Message size (0 is 20K value). This is a helper size, if the size is too small the internal storage is reallocated to the size of the message and another get operation is performed.	I	DEC	8	0
JSM MID	Message Id. (*BLANK or message id)	I/O	CHAR	24	
JSMCID	Correlation Id. (*BLANK or correlation id)	I/O	CHAR	24	
JSMRQMGR	Reply Queue Manager	O	CHAR	48	
JSMREPLYQ	Reply Queue	O	CHAR	48	
JSM MTTYPE	Message Type	O	DEC	9	0
JSMSTS	Status	O	CHAR	20	
JSMMSG	Message	O	CHAR	256	

When the LANSAs function or RPG program receives the Message Id and Correlation Id, do not send these values across to the loaded JSM service, because they do not contain valid EBCDIC characters.

The Message Id and Correlation Id can be used to get a message that matches one or both of these input values, if you require the next message set the message id and correlation id fields to *BLANK before calling the JSMMQGET program.

Example

```
CHANGE FIELD(#JSMQMGR) TO('MY.QMANAGER')
CHANGE FIELD(#JSMQNAME) TO('MY.QUEUE')
CHANGE FIELD(#JSMWAIT) TO(-1)
CHANGE FIELD(#JSMSIZE) TO(0)
```

CHANGE FIELD(#JSMMID) TO(*BLANK)
CHANGE FIELD(#JSMCID) TO(*BLANK)

CALL PGM(JSMMQGET) PARM(#JSMQMGR #JSMQNAME #JSMWAIT #

JSMMQPUT - Put message on queue

Parameter	Description	I/O	Type	Len	Dec
JSMQMGR	Queue Manager	I	CHAR	48	
JSMQNAME	Queue	I	CHAR	48	
JSMCCSID	Message CCSID (0 for queue default)	I	DEC	8	0
JSM PST	Persistence (Y N *BLANK for default)	I	CHAR	1	
JSMPT Y	Priority (0 to 9 -1 for queue default)	I	DEC	8	0
JSMMID	Message Id (*BLANK for new message id)	I	CHAR	24	
JSMCID	Correlation Id (*BLANK or value)	I	CHAR	24	
JSM MTYPE	Message Type 8 - MQMT_DATAGRAM 65536-999999999 - User defined	I	DEC	9	0
JSMRQMGR	Reply Queue Manager	I	CHAR	48	
JSMREPLYQ	Reply Queue	I	CHAR	48	
JSMSTS	Status	O	CHAR	20	
JSMMSG	Message	O	CHAR	256	
Optional					
JSMEXPIRY	Expiry, milliseconds, -1 for unlimited (default)	I	DEC	9	0

JSMUSER	User	I	CHAR	12	
---------	------	---	------	----	--

When you use the optional JSMEXPIRY parameter, you must also pass the JSMUSER parameter as well. Failing to pass the JSMUSER parameter will cause the JSMMQPUT operation to fail.

Example

```
CHANGE FIELD(#JSMCCSID) TO(0)
CHANGE FIELD(#JSMMTYPE) TO(8)
CHANGE FIELD(#JSM MID) TO(*BLANK)
CHANGE FIELD(#JSMCID) TO(*BLANK)
CHANGE FIELD(#JSMPST) TO(Y)
CHANGE FIELD(#JSMPTY) TO(4)
CHANGE FIELD(#JSMRQMGR) TO(*BLANK)
CHANGE FIELD(#JSMREPLYQ) TO(*BLANK)
```

```
CALL PGM(JSMMQPUT) PARM(#JSMQMGR #JSMQNAME #JSMCCSID
```

If the MQSeries MQCONN or MQPUT API call fails the current message is put to a recovery file JSMMQPUT if it exists. If the message length exceeds 32000 bytes the data is truncated.

The DDS source for physical file JSMMQPUT is in QDDSRC file in the JSM library.

```
DATE      8A   CCYYMMDD
TIME      6A   HHMMSS
JOBNUMBER 6A   000000
QUEUE     48A
MANAGER   48A
CCSID     8P 0
PERSIST   1A
PRIORITY  8P 0
USERID    12A
EXPIRY    9P 0
MESSAGEID 24A
```

CORRLATEID 24A
MSGTYPE 9P 0
RELYMGR 48A
RELYQUEUE 48A
MSGLENGTH 9P 0
MESSAGE 32000A

Also see

[MQSeries and DataQueue programs Example](#)

5.1.16 Data Queue Programs

All programs will return at least a status and message. The status is a string value or an i5/OS API reason code, such as:

- OK
- ERROR
- FATAL
- NOMSG
- CPF9801 - Object not found
- and so on...

JSMDQGET - Get message

Parameter	Description	I/O	Type	Len	Dec
JSMQLIB	Library	I	CHAR	10	
JSMQNME	Queue	I	CHAR	10	
JSMQKEY	Key - Use a blank key value for a non-keyed data queue.	I	CHAR	256	
JSMWAIT	Wait time (millisecs, rounded to nearest second)	I	DEC	8	0
JSMSTS	Status	O	CHAR	20	
JSMMSG	Message	O	CHAR	256	

Example

```
CHANGE FIELD(#JSMQLIB) TO(MYLIB)
CHANGE FIELD(#JSMQNME) TO(MYQUEUE)
CHANGE FIELD(#JSMQKEY) TO(*BLANK)
```

```
CALL PGM(JSMDQGET) PARM(#JSMQLIB #JSMQNME #JSMQKEY #JS
```

JSMDQPUT - Put message

--	--	--	--	--	--

Parameter	Description	I/O	Type	Len	Dec
JSMQLIB	Library	I	CHAR	10	
JSMQNME	Queue	I	CHAR	10	
JSMQKEY	Key - Use a blank key value for a non-keyed data queue.	I	CHAR	256	
JSMSTS	Status	O	CHAR	20	
JSMMSG	Message	O	CHAR	256	

Example

```
CALL PGM(JSMDQPUT) PARM(#JSMQLIB #JSMQNME #JSMQKEY #JS
```

Also see

[MQSeries and DataQueue programs Example](#)

5.1.17 IBM Toolbox for Java

JSM services such as the PDFSpoolFileService require IBM Toolbox for Java or JTOpen.

JSM ships with Open Source Software, JTOpen 8.2, codebase 5770-SS1 V7R2M0.00 built=20140313.

- The IBM Toolbox for Java is a library of Java classes that give Java programs easy access to IBM i data and resources.
- JTOpen is the open source version of Toolbox for Java.
- This toolbox is available as License Program 57xxJC1 - Toolbox for Java.
- This toolbox is available as License Program 57xxSS1 Extended Base Directory Support for V7R1.
- The jt400.jar file is located in the /QIBM/ProdData/HTTP/Public/jt400/lib directory.

For further information refer to:

[Toolbox for Java and JTOpen](#)

IBM provides two JDBC drivers for the IBM i database:

Native JDBC driver

This is shipped as part of the IBM i Developer Kit for Java (57xxJV1).

It is implemented by making native method calls to the SQL CLI (Call Level Interface).

Consequently, it only runs on the IBM i JVM.

The class name to register is com.ibm.db2.jdbc.app.DB2Driver. The URL subprotocol is db2.

Toolbox JDBC driver

This is shipped as part of the IBM Toolbox for Java.

It is implemented by making direct socket connections to the database host server.

This JDBC driver runs on any JVM.

The class name to register is com.ibm.as400.access.AS400JDBCdriver. The URL subprotocol is as400.

Physical file journaling for JDBC updates

```
CRTJRNRCV JRNRCV(MYLIB/JDBC) TEXT('JDBC Journal Receiver')
```

CRTJRN JRN(MYLIB/JDBC) JRNRCV(MYLIB/JDBC) TEXT('JDBC Journal')
STRJRNPF FILE(MYLIB/MYFILE) JRN(MYLIB/JDBC) IMAGES(*BOTH)

It is easy to forget to start journaling when you create or change a physical file. You can avoid such problems by creating a data area named QDFTJRN in any library where you want to automatically start journaling.

The new data area contains the name of a journal, and as long as the person creating the physical file has adequate authority, journaling will be started to the journal named in the data area.

You specify the journal library in the first 10 positions, and the second 10 positions contain the journal, and the last five positions contain *FILE. You can also specify *NONE in the last five positions to prevent journaling from starting.

5.1.18 Common Command Keywords

ARCHIVE	FRAGMENT	SVMODE
BIND	LOCALE	SVQUOTE
CONTENT	METHOD	SVROW
DOMGET	NUMBERFORMAT	SVROWLIMIT
DOMGETRESULT	OUTPUT	SVTABLE
DOMSET	SCHEMA	TRIM
DOMSETMODEL	SEPARATOR	TRUNCATE
DOMSETRESULT	SVCOLUMN	VALIDATING
ENCODING	SVHEAD	VERSION
FILTER	SVLABEL	XSL

ARCHIVE

The optional keyword **ARCHIVE** is used to archive content to a specified path. The content encoding of the archive can be controlled by the use of an **ARCENCODE** keyword. If this keyword is not present then the 'archive.encoding' service property entry is used, if this service property does not exist then a default encoding of UTF-8 is used. To use the default file encoding of the JVM, use the keyword value of ***DEFAULT**.

```
ARCHIVE(/MYARCHIVE/FILE.DAT)
ARCHIVE(/MYARCHIVE/FILE.DAT) ARCENCODE(*DEFAULT)
ARCHIVE(/MYARCHIVE/FILE.DAT) ARCENCODE(ISO8859_1)
```

```
#
# archive.encoding=ISO8859_1
archive.encoding=*DEFAULT
#
```

BIND

The optional keyword BIND can be used by HTTP service's SERVICE_LOAD command to automatically bind the optional query string parameters to the function fields or to fill a working list.

The command must contain a SERVICE_EXCHANGE(*FIELD) keyword or a working list to allow data binding to occur.

The query string parameters need to be supplied in a standard format:

```
/cgi-bin/jsmdirect?orderxml+fieldname(fieldvalue)+fieldname(fieldvalue)+...
```

```
/cgi-bin/jsmdirect?orderxml+period(12)+id(acme)
```

If the BIND keyword contains the special value of *FIELD, this will cause the function fields to be set with the field value.

If the BIND keyword contains the special value of *LIST, this will cause the column 1 of the working list argument to receive the field name and column 2 to receive the field value. The working list columns can have any name and size.

Example

```
SERVICE_LOAD SERVICE(HTTPInboundXMLService) BIND(*FIELD) SE
```

```
SERVICE_LOAD SERVICE(HTTPInboundXMLService) BIND(*LIST) SER
```

CONTENT

The optional keyword CONTENT is used by HTTP services to control the type of content being accepted or the content type being sent.

It is highly recommended not to use the CONTENT keyword and accept the default value.

All HTTP services and content handlers are expecting a particular content type, if content is being received and its content type does not match the content, then use the CONTENT keyword to tell the service to accept the content if the content type matches the keyword value.

Some Outbound services and content handlers use the CONTENT keyword to specify the type of content to be created. The SEND command available with HTTPInboundSVService service and InboundSeparatedValue handler use the CONTENT keyword to direct the service to create CSV or TSV content.

A CONTENT keyword value of *ANY allows the service or content handler to receive content with any content type.

It is recommended to use the short cut names for the content keyword value.

*XML	application/xml
*TEXTXML	text/xml
*TEXTPLAIN	text/plain
*SOAP	application/soap+xml
*HTML	text/html
*XHTML	application/xhtml+xml
*CSV	application/comma-separated-values
*TEXTCSV	text/ x-comma-separated-values
*TSV	application/comma-separated-values
*TEXTTSV	text/ x-tab-separated-values
*X12	application/edi-x12
*EDIFACT	application/edifact

*STREAM	application/octet-stream
*PDF	application/pdf
*ZIP	application/zip
*JSON	application/json
*EXCEL	application/vnd.ms-excel
*CRL	application/pkix-crl
*GIF	image/gif
*PNG	image/png
*JPEG	image/jpeg
*SVG	image/svg+xml
*MPEG	audio/mpeg
*MPEG4	video/mpeg4

Example

```
SERVICE_LOAD SERVICE(HTTPInboundXMLService) CONTENT(*HTML)
```

```
SERVICE_LOAD SERVICE(HTTPInboundXMLService) CONTENT(*ANY)
```

ENCODING

The optional keyword `ENCODING` is used to specify what encoding must be applied to a byte content to convert it to a Unicode string.

An `ENCODING` keyword value of `*DEFAULT` can be used to select the default encoding for the JVM.

XML services support the encoding value of `*AUTODETECT`. auto-detection of encoding is only possible on XML content and cannot be used for reading other content like comma-separated text files.

For HTTP services the `ENCODING` on the `SERVICE_LOAD` command enables the service to convert the standard input byte stream.

If no `ENCODING` value is available, the service uses the HTTP protocol content type charset value to determine the encoding used.

If no charset is present the encoding defaults to the JVM default file encoding or auto-detect for XML content.

The `ENCODING` value can be used on the `SEND` command of HTTP services to encode the HTTP request/response content, an appropriate charset value is automatically append to the end of the content-type protocol line. The default encoding is UTF-8.

If the HTTP server is an IBM IBM i in `MIXED` output mode and the content type selected starts with 'text/', the encoding will have no effect, the text content will be encoded using the `CGI-EBCDIC-CCSID` value and the HTTP server will set the charset based on the `DefaultNetCCSID` entry in the server configuration.

Note that you need to run the International version of Java run-time. The US-only version does not contain all the encodings.

For further information refer to [5.1.2 IANA Encodings](#).

Example

```
SERVICE_LOAD SERVICE(HTTPInboundXMLService) ENCODING(ISO88591)
```

METHOD

The optional keyword **METHOD** is used to specify the HTTP request method to be used for outbound HTTP services. The default value is **POST**. The HTTP protocol standard defines several methods and user-defined methods can be used for REST style applications.

For more information refer to:

[Method Definitions](#)

METHOD	GET	Retrieve resource.
	PUT	Replace resource.
	POST	Update resource.
	HEAD	Get retrieve resource header.
	TRACE	Trace access to resource.
	DELETE	Delete resource.
	OPTIONS	Get communication options.
	user-defined	

VERSION

The optional keyword **VERSION** is used to specify the HTTP request protocol version. The default is 1.1.

For the SOAPAgentService the default is 1.0.

VERSION 0 HTTP/1.0

1.0 HTTP/1.0

1 HTTP/1.1

1.1 HTTP/1.1

XSL

The mandatory keyword XSL is used to specify which XSLT file will be used for transformation.

Inbound XML transactions use the XSLT to transform XML document into FunctionXML format which is used by the service to bind the XML data to specified fields and working list.

Outbound XML transactions use the XSLT to transform FunctionXML to the required flavor of XML.

The XSL keyword value is used in a service properties resource lookup to determine the location of the XSLT file.

The first resource lookup is for a possible Translet class name using the xslc. {value}, if this entry is not present, then a second lookup is done using the xsl. {value} for file name.

Example

```
RECEIVE XSL(ORDER) SERVICE_LIST(...)
```

```
# xsl.order=xsl/receive-order.xsl
```

```
xslc.order=com.lansa.jsm.translet.ReceiveOrder
```

```
#
```

SCHEMA

The optional keyword SCHEMA is used to turn on XML schema validation. By default, XML schema validation is disabled. XML Schema validation can also be controlled by the service property 'validation.schema'.

Example

```
SERVICE_LOAD SERVICE(HTTPInboundXMLService) SCHEMA(*YES)
```

```
#
```

```
validation.schema=*yes
```

```
#
```

VALIDATING

The optional keyword `VALIDATING` is used to configure the service to use a validating or non-validating XML parser.

The default is to use a validating XML parser. This option can also be controlled by the service property `'validation.parser'`.

A nonvalidating parser ensures that the XML data is well formed, but does not verify that it is valid.

A validating parser uses the XML document defined DTD or XMLSchema grammars to validate that the XML data elements and attributes conform to the structural constraints of these schemas.

Why run in nonvalidating mode when a parser is capable of validation?.

Because validation can significantly impact performance, especially when long and complex DTDs or XMLSchemas are involved.

Some developers find that while enabling validation during development and test phases is crucial, it's sometimes beneficial to surpress validation in production systems where document throughput is most valued and the reliability of the data is already known.

Example

```
SERVICE_LOAD SERVICE(HTTPInboundXMLService) VALIDATING(*NO)

#
validation.parser=*no
#
```

FRAGMENT

A fragment is a partial segment of a complex data object. Fragments are used to build complex data structures, such as XML, using individual LANSAs fields and single working lists. They are also used to convert incoming complex structures back into fields and working lists.

The optional keyword FRAGMENT can be used by content handlers and services that process XML, to help in the creation of outbound XML.

The FRAGMENT keyword value specifies the storage name for the created XML.

Internally, FunctionXML is created using the field list and list objects. This FunctionXML is then transformed using the supplied XSL into the desired flavor of XML.

If the resultant XML contains an <rdml:fragment /> tag, then a stored XML fragment under this name is merged into the resultant XML. If a FRAGMENT keyword is present, then the resultant XML is stored under the fragment name for later merging.

If no FRAGMENT keyword is present, the resultant XML becomes the final output XML message.

The FRAGMENT keyword is also used by the [Soap Agent Service](#) to assist in the creation of outgoing SOAP objects, and in retrieving the data in a SOAP response. Use the SOAP Wizard to specify which parts of the SOAP object represent fragments, and to give each fragment a name.

For XML services, if the OUTPUT option is *ADD then the fragment is concatenated to the fragment already stored under that name. The *NEW option will create a stored fragment if one does not exist or replace an existing one.

Example

```
SEND XSL(ORDER_DETAIL) FRAGMENT(DETAIL) SERVICE_LIST(...)
```

OUTPUT

The optional keyword OUTPUT is used to append XML fragments.

If no OUTPUT keyword is present, then the value *NEW is used.

The OUTPUT keyword value *ADD directs the service to append the created XML fragment to the end of the previously created XML fragment of the same storage name. If no previous storage exists, then a new one is created.

The OUTPUT keyword value *NEW directs the service to create a new fragment storage if one does not exist or replace an existing fragment stored under the specified fragment name.

NUMBERFORMAT

The optional keyword NUMBERFORMAT is used to handle numeric strings, where the decimal separator is not the decimal point character ".".

XML content always uses the "." as a decimal separator, but CSV or TSV data could use another character.

The keyword value can take one of the special values or a locale string.

The presence of this keyword will override other default actions.

NUMBERFORMAT	*NONE	No formatting.
	*DEFAULT	Use the JDK default locale.
	*CLIENT	Use the locale of the LANSAs function or RPG program job.
	*USERAGENT	Use the locale of the remote user agent.
	value	Locale string.

If no keyword is present, the 'numberformat.locale' entry in the associated service's properties file is used.

If there is no keyword or 'numberformat.locale' no formatting is done.

For inbound content handlers if the number format is *USERAGENT, the HTTP protocol property X-USER-AGENT-LOCALE is used, if this is not available then the *CLIENT value is used.

For outbound content handlers if the number format is *USERAGENT, the value of the LOCALE keyword is used.

Example

```
RECEIVE NUMBERFORMAT(EN_AU)

#
numberformat.locale=EN_AU
#
```

LOCALE

The optional keyword **LOCALE** is used by services that receive and send content to the LANSAs Integrator User Agent desktop application, or other LANSAs Integrator services.

Outbound content handlers can use this keyword to control the value of the **X-USER-AGENT-LOCALE** HTTP protocol property.

If the keyword value is a `language_country` string value, this is converted into a Locale object.

If the keyword value is `*DEFAULT`, the locale returned from `Locale.getDefault()` method is used.

If the keyword value is `*CLIENT`, the locale returned from the `command.getClient().getLocale()` method is used.

If no **LOCALE** keyword is present, then the value of `*CLIENT` is used.

Example

```
SEND LOCALE(EN_AU)
```

```
SEND LOCALE(*CLIENT)
```

SVROW

The optional keyword `SVROW` is used to specify the starting row when reading records.

The default value is 1.

SVROWLIMIT

The optional keyword SVROWLIMIT is used to specify the number of rows to be read.

SVROWLIMIT *NONE	Default. There is no limit, read all records.
*LIST	The number of records read is equal to the max entry count of the working list.
*AVAILABLE	The number of records read is equal to the max entry count minus the number of current entries in the working list.
value	An integer value.

SVMODE

The optional keyword SVMODE is used by content handlers and services that process separated value data to control how to handle the inbound separated value data.

- SVMODE *NONE Default. Inbound separated value data does not contain a field header record.
- *IGNORE Inbound separated value data contains a field header record. This record is to be ignored and not included in the received data list.
- *USE Inbound separated value data contains a field header record. This record is to be used in data column field mapping.

If the SVHEAD keyword is not present and the SVMODE keyword is *NONE or *IGNORE then the inbound data is inserted directly in the working list, so the field count and data types must match the list definition.

Example

```
RECEIVE SVMODE(*IGNORE) SERVICE_LIST(...)
```

SVQUOTE

The optional keyword SVQUOTE is used to explicitly double quote text values or all values.

The possible values are *ALL, *TEXT or *NONE which is also the default value.

A value of *ALL means that all values are double quoted.

A value of *TEXT means only text values are double quoted.

SVHEAD

The optional keyword SVHEAD is used to describe the field layout of the separated value data.

Inbound

The SVHEAD keyword has different roles for inbound separated value data and outbound separated value data.

When the SVHEAD keyword is used on the RECEIVE or READ command for processing inbound data, the following rules apply.

If the SVHEAD keyword is not present and the SVMODE keyword is *NONE or *IGNORE then the inbound data is inserted directly in the working list, so the field count and data types must match the list definition.

If the SVHEAD keyword is present, then a look up on the service properties resource is done using the sv.head.{value} to determine the header record that describes the column layout of the inbound data.

This header record is used to map column value to working list field for each record as its being added to the list.

Example SVFileService.properties

```
sv.head.order=LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY
```

Example

```
READ SVHEAD(ORDER) SERVICE_LIST(...)
```

```
RECEIVE SVHEAD(ORDER) SERVICE_LIST(...)
```

Outbound

To add a header record to outbound separated value data, the SVHEAD keyword can be used on the SEND or WRITE command.

The keyword value is used to locate the header record using the sv.head.{value} lookup on the service properties resource.

A keyword value of *LIST will cause the service to use the working list fields.

A keyword value of *COLUMN will cause the service to use the fields from the SVCOLUMN lookup.

Example

```
WRITE SVHEAD(*LIST) SERVICE_LIST(...)
```

```
SEND SVHEAD(*LIST) SERVICE_LIST(...)
```

SVEXCLUDE

The keyword SVEXCLUDE is used to exclude working list fields from the content being created by the WRITE command.

SVCOLUMN

The keyword SVCOLUMN is used to define the relational database columns, a look up on the service properties resource is done using the sv.column.{value}.

The SVTABLE keyword is used to define the database table.

Example SVFileService.properties

```
sv.column.order=LINE_NUMBER,PART_NUMBER,PART_DESC,PART_AN
```

SVTABLE

The keyword SVTABLE is used to identify the relational database table that is used for the SQL select or insert operations.

The SVCOLUMN keyword is used to define the table columns.

Example

```
READ FILE(order.csv) SVTABLE(NEW_ORDERS) SVCOLUMN(ORDER)
```

Example SVFileService.properties

```
sv.column.order=LINE_NUMBER,PART_NUMBER,PART_DESC,PART_AM
```

SVLABEL

The optional keyword SVLABEL is used to include the keyword value as the first column value for database table inserts or as the where constraint for database table selects.

Example

1. In the following example, the value 23 is inserted into the first column (ID).

```
READ FILE(order.csv) SVTABLE(ORD) SVCOLUMN(ORDER) SVLABI
"INSERT INTO ORD(ID,LINENUM,PARTNUM,PARTDSC,PARTAMT,PA
```

2. In the following example, the first column (ID) is used to receive SV values.

```
READ FILE(order.csv) SVTABLE(ORD) SVCOLUMN(ORDER)
"INSERT INTO ORD(ID,LINENUM,PARTNUM,PARTDSC,PARTAMT,PA
```

3. In the following example, the ID column is not in the selection but is used in

```
WRITE FILE(order.csv) SVTABLE(ORD) SVCOLUMN(ORDER) SVLAE
"SELECT LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY FROM
```

4. In the following example, the ID column is included in the selection, because

```
WRITE FILE(order.csv) SVTABLE(ORD) SVCOLUMN(ORDER)
"SELECT ID,LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY FR
```

SVFileService.properties

```
sv.column.order=ID,LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY
```

SEPARATOR

The keyword SEPARATOR is used by separated value services to identify the content separator.

SEPARATOR value	Separator character.
-----------------	----------------------

*COMMA	, Default.
--------	------------

*SEMICOLON	;
------------	---

*TAB	
------	--

*TILDE	~
--------	---

Example

```
READ SEPARATOR(~)
```

```
READ SEPARATOR(*TILDE)
```

TRIM

The optional keyword TRIM allows the service to trim blanks from data values. Data values coming from the function to the service and visa versa are trimmed. The default value for the TRIM keyword is *TRAILING.

Note: TRIM is implemented at the command level and not at the service level. This means that all services have TRIM capability.

- TRIM *NONE No trimming.
- *BOTH Trim leading and trailing blanks.
- *LEADING Trim leading blanks.
- *TRAILING Trim trailing blanks.

Example

```
SEND TRIM(*BOTH)
```

```
RECEIVE TRIM(*NONE)
```

TRUNCATE

The optional keyword TRUNCATE allows the service to truncate data values.

Only data values coming from the service to the function are truncated.

The default value for the TRUNCATE keyword is *NONE.

Note: TRUNCATE is implemented at the command level and not at the service level. This means that all services have TRUNCATE capability.

TRUNCATE *NONE No truncating.

*LIST Truncate list values.

*FIELD Truncate field values.

*BOTH Truncate field and list values.

Example

```
RECEIVE TRUNCATE(*LIST)
```

DOMSET

The optional keyword **DOMSET** allows the service to read the input XML source directly, or to pre-process the XML data into an ideal format for the Apache XML parser.

The default value for the **DOMSET** keyword is ***READER** to maintain compatibility.

The recommended value is ***STREAM**.

The **DOMSET**, **DOMSETMODEL** and **DOMSETRESULT** settings are used by commands like **RECEIVE** for inbound XML processing.

DOMSET *STREAM Read XML directly.

***READER** Process XML into an ideal form and pass to XML parser as a `StringReader`.

Example

```
SERVICE_LOAD DOMSETMODEL(*DOCUMENT) DOMSET(*STREAM)
```

DOMSETMODEL

The optional keyword DOMSETMODEL specifies how the input XML source will be stored.

The default value for the DOMSETMODEL keyword is *STRING to maintain compatibility.

If you plan to use translets (Compiled Stylesheets) to read XML content you can only use the DOMSETMODEL(*STRING) option.

The recommended value is *DOCUMENT.

The DOMSET, DOMSETMODEL and DOMSETRESULT settings are used by commands like RECEIVE for inbound XML processing.

DOMSETMODEL *DOCUMENT Create XML DOM.

*STRING Create XML String.

Example

```
SERVICE_LOAD DOMSETMODEL(*DOCUMENT) DOMSET(*STREAM,
```

DOMSETRESULT

The optional keyword DOMSETRESULT specifies how the style sheet transformation result will be stored.

If the transformation result from input XML source to FunctionXML is going to be large, then use *FILE so the transformation result is streamed out to a temporary file. This will greatly reduce the amount of memory used by the transformation process, as the result is not stored in memory.

The default value for the DOMSETRESULT keyword is *UTF8.

The DOMSET, DOMSETMODEL and DOMSETRESULT settings are used by commands like RECEIVE for Inbound XML processing.

DOMSETRESULT *UTF8 XML transform is kept as an array UTF-8 encoded bytes.

*STRING XML transform is kept as a String object.

*FILE XML transform is steamed out to a temporary file.

Example

```
SERVICE_LOAD DOMSETMODEL(*DOCUMENT) DOMSET(*STREAM,
```

DOMGET

The optional keyword DOMGET specifies how FunctionXML created from the LANSAs function fields and working list will be stored.

If the FunctionXML is going to be large, then use *FILE so the input FunctionXML source resides in a temporary file. This will greatly reduce the amount of memory used by the transformation process, as the input FunctionXML source is not stored in memory.

The default value for the DOMGETRESULT keyword is *STRING to maintain compatibility.

If you plan to use translets (Compiled Stylesheets) to create XML content you can only use the DOMGET(*STRING) option.

The DOMGET and DOMGETRESULT setting are used by commands like SEND for Outbound XML processing.

DOMGET *UTF8 FunctionXML is kept as UTF-8 encoded bytes.

*STRING FunctionXML is kept as a String object.

*FILE FunctionXML is steamed out to a temporary file.

Example

```
SERVICE_LOAD DOMGET(*FILE) DOMGETRESULT(*FILE)
```

DOMGETRESULT

The optional keyword `DOMGETRESULT` specifies how the style sheet transformation result will be stored.

The transformation process requires the input `FunctionXML` source to be read into an in-memory XML DOM object and the style sheet to be applied to this DOM.

Currently due to outbound fragment handling requirements, the XML result is read and converted to a `String` for later fragment merging.

Streaming the result out to a file is still valid, as only memory for the DOM is required for this type of transform and the result file content to `String` conversion is done at a later stage.

The default value for the `DOMGETRESULT` keyword is `*STRING`.

The `DOMGET` and `DOMGETRESULT` setting are used by commands like `SEND` for Outbound XML processing.

`DOMGETRESULT *UTF8` XML transform is kept as UTF-8 encoded bytes.

`*STRING` XML transform is kept as a `String` object.

`*FILE` XML transform is steamed out to a temporary file.

Example

```
SERVICE_LOAD DOMGET(*FILE) DOMGETRESULT(*FILE)
```

FILTER

The **FILTER** or **FILTERCLASS** keyword is used to specify the content filter class to be applied to the XML DOM document. The **FILTER** keyword value is used to lookup the filter classname specified by the `filter.'value'` property in the `XMLBindFileService` properties file. The filter classname can also be directly specified using the **FILTERCLASS** keyword.

Example

```
package com.acme.filter ;

import java.io.* ;

import java.util.HashMap ;

import org.w3c.dom.Document ;

import com.lansa.jsm.JSMTrace ;
import com.lansa.jsm.JSMCommand ;
import com.lansa.jsm.JSMResource ;

public class SampleFilter implements com.lansa.jsm.service.ContentFilter
{
    public Object filter ( HashMap properties, Object content )
    {
        if ( content == null )
        {
            return null ;
        }

        if ( !(content instanceof Document) )
        {
            throw new IllegalArgumentException ( "SampleFilter: unexpected cont
        }

        JSMTrace trace = (JSMTrace)properties.get ( "jsm.trace" ) ;

        JSMCommand command = (JSMCommand)properties.get ( "jsm.commar
```

```
JSMResource resource = (JSMResource)properties.get ( "jsm.resource" )

if ( trace != null )
{
    m_trace.println ( "Apply SampleFilter" ) ;
}

/*
    Modify or create new object
*/

/*
    Return replacement or existing object
*/

return content ;
}
}
```

5.2 FTPService

The FTPService allows an application to send and receive files to or from an FTP (file transfer protocol) server. FTP is a TCP based service, widely used on the Internet and to support e-Commerce.

The FTPService supports standard and secure mode FTP. It provides a rich set of service commands for:

- Establishing a connection with a FTP server.
- Creating, deleting, renaming and navigating folders on the FTP server.
- Creating and deleting files
- Transferring files to or from the FTP server.

Related Services

The FTPService is not dependent on other services. Using just this one service a user can access and transfer files, from and to, another host over a network (assuming FTP is supported between your local and remote servers).

However, depending on the requirements of your application, you may wish, for example, to create a PDF file then distribute this file using the FTP service.

Technical Specifications

- FTP utilizes two ports; a 'data' port and a 'control' port (also referred to as the command port).

When your local FTP server first connects to the remote FTP server, it establishes a 'control session', to allow the two servers to give each other commands (e.g. change directories, get a file etc...). To initiate this connection your local server has to have a port of entry into the remote FTP server, like a door for your local server to knock on and the remote FTP server to open. The remote server probably has several entry ports, but most servers set aside Port 21 to receive connections from FTP clients.

In order for your local server to transfer files, a second session is established between your computer and the remote FTP server, the 'data session'. In normal FTP, the remote FTP server initiates this second connection using another port (typically Port 20).

5.2.1 What can I use the FTPService for?

FTP is commonly used to transfer or copy files from one computer to another on TCP/IP networks such as the internet. This includes the movement of files between machines with widely different operating systems. Any transfer of information between computers is subject to appropriate security on both the sending and receiving computers.

5.2.2 Using the FTPService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application to put a file to a remote FTP server would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

CONNECT

LOGIN

CHGDIR

BINARY

PUT

QUIT

SERVICE_UNLOAD

JSM(X)_CLOSE

The steps to get a file from a remote FTP server would be very similar, but uses the GET command instead of a PUT.

Refer to [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.2.3 FTPService Commands

Your application issues commands to the FTPService by passing the command strings through the Java Service Manager using the JSM_COMMAND or JSMX_COMMAND Built-In Function, or an API for your chosen development language.

The commands that the FTPService processes are:

SERVICE_LOAD

CONNECT

LOGIN

CHGDIR

GETDIR

ASCII

BINARY

NOOP

PUT

GET

SITE

QUOTE

CREATE

DELETE

RENAME

LIST

QUIT

SERVICE_GET

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
---->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

- | | |
|---------|--|
| SERVICE | The name of the service to be loaded - in this case FTPService. |
| TRACE | To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.
The possible values for the TRACE keyword are: <ul style="list-style-type: none">● *NO● *YES● *ERROR The optional TRACE_trace option *ERROR will turn on |

tracing and if the service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.

TRACE_NAME The optional TRACE_NAME keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVICE_LOAD')
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMCMD)
```

RDMLX

```
#jasmcommand := 'service_load service(FTPService)'
use builtin(jsmx_command) with_args(#jasmxhdl1 #jasmcommand) to_get(#jasmcommand)
```

CONNECT

The CONNECT command starts a connection to the remote FTP server.

```

                                     Required
CONNECT ----- HOST ----- host:port -----
>

                                     Optional
>-- SECURE ----- *NO ----->
      *YES
      *IMPLICIT
>-- PROTECTION ----- *NO ----->
      *YES
>-- PORT_ADDRESS ----- address ----->
>-- ENCODING ----- encoding ----->
>-- TIMEOUT ----- milliseconds -----|
```

Keywords

HOST	Nominate a FTP server to connect to. The FTP server can be specified as an IP address, nnn.nnn.nnn.nnn:port, or a domain name. If a port number is not supplied on an IP address the default value 21 is used unless the keyword SECURE (*IMPLICIT) is used in which case the default port value is 990.
SECURE	Indicates whether a secure control channel is required to protect FTP command information like user and password. The default value *NO indicates a plain data connection is to be used.

A value of *YES initially connects to the remote FTP server using a plain socket then changes to SSL/TLS connection using the AUTH subcommand.

A value of *IMPLICIT indicates a secure control channel directly to the SSL/TLS FTP server is required.

- PROTECTION** The default option *NO indicates a clear data channel is required.
The alternative value, *YES, indicates the data channel is to be secure.
- PORT_ADDRESS** The local IP address used by the PORT subcommand.
This keyword is not typically used but is available to indicate the IP address has been dynamically allocated, such as a PPP dialup connect after the JSM has been started.
- ENCODING** The remote server path encoding. The default option is *DEFAULT.
- TIMEOUT** The host connection timeout in milliseconds. The default is 0, which is an infinite timeout or an operating system timeout.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('CONNECT HOST(ISERII
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 'connect host(iseries01)') to
```

LOGIN

The LOGIN command authenticates the connection to the remote FTP server.

Required

```
LOGIN ----- USER ----- user ----->  
>-- PASSWORD ----- password -----|
```

Keywords

USER The user profile to be used to log into the remote FTP server.

PASSWORD The password corresponding to the value specified in the
USER keyword.

Comments / Warnings

The USER and PASSWORD values may be case sensitive depending on the FTP server.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('LOGIN USER(ftpuser) PA
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'login user(ftpuser) passwo
```

CHGDIR

The CHGDIR command will change the current directory on the remote FTP session.

```
CHGDIR ----- PATH ----- path name -----|
```

Keywords

PATH Changes the current directory path to the PATH value specified.
This value can be a relative or absolute path.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('CHGDIR PATH(/xxxpgml
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 'chkdir path(/xxxpgmlib/jsr
```

GETDIR

The GETDIR command will return the current working directory on the remote FTP session into the JSM command's message field.

```
GETDIR ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GETDIR') TO_GET(#JSM
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'getdir') to_get(#jsmxsts #js
```

ASCII

The ASCII command sets the session mode to ASCII.

Depending on the FTP server, ASCII mode allows the FTP server to convert the transmitted data to a format suitable for the host file system. CRLF characters could be converted to LF characters if the host is a Linux server. ASCII data could be converted to EBCDIC data if the host is an IBM i server.

```
ASCII ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('ASCII') TO_GET(#JSMS'
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'ASCII') to_get(#jsmxsts #j
```

BINARY

The BINARY command sets the session mode to BINARY.

Binary means that the contents of the transmission are preserved and this is the preferred mode.

```
BINARY ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('BINARY') TO_GET(#JSM
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'binary') to_get(#jsmxsts #j
```

NOOP

The NOOP command is a NO OPERATION command. The remote FTP server should return an OK status.

```
NOOP ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('NOOP') TO_GET(#JSMS'
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhde1 'noop') to_get(#jsmxsts #jsi
```

PUT

The PUT command will send the contents of a local file to the remote FTP server.

The file specified by the FROM keyword is sent to the remote FTP server, using the DATALINK mode. The remote FTP server will store the file using the path specified by the TO keyword.

```
PUT ----- FROM ----- file path ----->
    >-- TO ----- file path ----->
                Optional
    >-- FROM_ENCODING ---- value -----
>
    >-- TO_ENCODING ----- value ----->
    >-- CONVERT ----- *NO ----->
                *YES
    >-- RECORD_SIZE ----- 1024 ----->
                *NOMAX
                integer
    >-- RECORD_DELIMITER - *NONE -----
--->
                *LF
                *CRLF
    >-- DATALINK ----- *PASV -----|
                *PORT
```

Keywords

FROM

Nominate the local file path to be copied to the

remote FTP server. This value can be a relative or absolute path. If the path is relative, the current working directory is the JSM instance directory.

See Comments / Warnings below.

TO	<p>Nominate the remote file path where the file is to be placed on the remote FTP server. This value can be a relative or absolute path. If the path is relative, the current working directory is the remote FTP server's current directory.</p> <p>See also GETDIR command.</p>
FROM_ENCODING	<p>Encoding is used to convert the FROM file content to an intermediate Unicode string.</p> <p>A default value is obtained from the 'from.encoding' property resource. If no property resource is located the value defaults to Cp037.</p> <p>Only use this keyword when CONVERT is set as *YES.</p>
TO_ENCODING	<p>Encoding is used to convert the intermediate Unicode string to target encoding.</p> <p>A default value is obtained from the 'to.encoding' property resource. If no property resource is located the value defaults to ISO8859_1.</p> <p>Only use this keyword when CONVERT is set as *YES.</p>
CONVERT	<p>The default conversion value is *NO.</p> <p>A value of *YES indicates that after each record size read, the data will be converted to the specified target encoding and sent to the FTP server.</p>
RECORD_SIZE	<p>The default value is 1024 bytes.</p> <p>A value of *NOMAX sets the read size to the length of the source file.</p> <p>Nominating an integer value specifies the read size, in bytes, to be used on the source file.</p> <p>Refer to Comments / Warnings below.</p>

RECORD_DELIMITER The default value is *NONE.

A value is only required when keyword CONVERT is set as *YES. In this case the end of record marker to be added to the end of each converted record read can be nominated as either *LF (Line Feed) or *CRLF (Carriage Return, Line Feed).

For more information refer to [5.1.7 Carriage Return, Line Feed and New Line](#).

DATALINK

The DATALINK keyword determines how the "control" session is initiated. The default value is *PASV.

In passive mode, *PASV, the FTP the client initiates both connections to the remote FTP server. Passive mode causes the FTPService to send a PASV subcommand to the remote FTP server, the FTP server returns the port number of the socket server it has started and a data transfer channel is established from the local server using this port number. This is known as "Passive FTP" and is often used to get around firewall difficulties. Using "Passive" FTP the local server establishes the second connection, not the remote FTP server. Most firewalls will allow your computer to transfer files in this second connection to the remote FTP server (just as it allowed your computer to establish the first connection).

Port mode, *PORT, causes the FTPService to start a socket server and informs the remote FTP server via the PORT subcommand of the port number this socket server is listening on, the remote FTP server then connects to this socket server and a data transfer channel is established.

Refer to [Technical Specification](#) for more information.

Comments / Warnings

If the FROM file is an IBM i physical file:

- The RECORD_SIZE must match the total record length of the file.
- Use the DSPFD command to determine the total record length.
- All fields in the file should be of type CHAR.
- A deleted record in the file will be read as a blank record, so use the RGZPFM command to removed deleted records from the file.

Normal stream files can also be converted:

- After each record size read the data is converted.
- The FROM file content must be single byte encoded, unless the RECORD_SIZE (*NOMAX) option is used, in which case the file is processed in one read and the conversion is applied to the entire file content.
- For more information refer to [5.1.7 Carriage Return, Line Feed and New Line](#).

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('PUT FROM(order.xml) TO(/xmldata/orde  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

or

```
CHANGE FIELD(#JSMCMD) TO('PUT FROM(order.xml) TO(/xmldata/orde  
8)')  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

or

```
CHANGE FIELD(#JSMCMD) TO('PUT FROM(order.xml) TO(/xmldata/orde  
8)')  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

From an IBM i file to a text file:

```
CHANGE FIELD(#JSMCMD) TO('PUT FROM(/QSYS.LIB/JSMLIB.LIB/DC  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jsmcmd := 'put from(' + #jsmfrom + ') to(' + #jsmto + ')'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

GET

The GET command will receive the contents of a remote file and save it to a local file. The file specified by the FROM keyword is retrieved from the remote FTP server, using the DATALINK mode. The local server will store the file using the path specified by the TO keyword.

```
GET ----- FROM ----- file path ----->
    >-- TO ----- file path ----->
                Optional
    >-- FROM_ENCODING ----- value -----
>
    >-- TO_ENCODING ----- value ----->
    >-- CONVERT ----- *NO ----->
                *YES
    >-- RECORD_SIZE ----- 1024 ----->
                integer
    >-- RECORD_DELIMITER ----- *NONE -----
--->
                *LF
                *CRLF
    >-- DATALINK ----- *PASV -----|
```

Keywords

- | | |
|------|--|
| FROM | Nominate the remote file path to be copied from the remote FTP server.
See also CHGDIR command. |
| TO | Nominate the local file path where the file is to be placed on the remote FTP server. |

See Comments / Warnings below.

FROM_ENCODING Encoding is used to convert the FROM file content to an intermediate Unicode string.
A default value is obtained from the 'from.encoding' property resource. If no property resource is located the value defaults to Cp037.
Only used this keyword when CONVERT is set as *YES.

TO_ENCODING Encoding is used to convert the intermediate Unicode string to target encoding.
A default value is obtained from the 'to.encoding' property resource. If no property resource is located the value defaults to ISO8859_1.
Only used this keyword when CONVERT is set as *YES.

CONVERT The default is *NO conversion.
A value of *YES indicates that after each record size read, the data will be converted to the specified target encoding and sent to the FTP server.

RECORD_SIZE The default value is 1024 bytes.
Nominating an integer value specifies the read size used on the source file.
Refer to [PUT Comments / Warnings](#).

RECORD_DELIMITER The default value is *NONE.
A value is only required when keyword CONVERT is set as *YES. In this case the end of record marker to be added to the end of each converted record read can be nominated as either *LF (Line Feed) or *CRLF (Carriage Return, Line Feed).
For more information refer to [5.1.7 Carriage Return, Line Feed and New Line](#)

DATALINK The DATALINK keyword determines how the "control" session is initiated. The default value is

***PASV.**

In passive mode, ***PASV**, the FTP the client initiates both connections to the server. Passive mode causes the FTPService to send a PASV subcommand to the remote FTP server, the FTP server returns the port number of the socket server it has started and a data transfer channel is established from the local server using this port number. This is known as "Passive FTP" and is often used to get around firewall difficulties. Using "Passive" FTP, the local server establishes the second connection, not the remote FTP server. Most firewalls will allow your computer to transfer files in this second connection to the remote FTP server (just as it allowed your computer to establish the first connection).

Port mode, ***PORT**, causes the FTPService to start a socket server and informs the remote FTP server via the PORT subcommand of the port number this socket server is listening on, the remote FTP server then connects to this socket server and a data transfer channel is established.

Refer to [Technical Specifications](#) for more information.

Comments / Warnings

If the FROM file is an IBM i physical file:

- The RECORD_SIZE must match the total record length of the file.
- Use the DSPFD command to determine the total record length.
- All fields in the file should be of type CHAR.
- A deleted record in the file will be read as a blank record, so use the RGZPFM command to removed deleted records from the file.

Normal stream files can also be converted:

- After each record size read the data is converted.
- The FROM file content must be single byte encoded, unless the RECORD_SIZE (*NOMAX) option is used, in which case the file is processed in one read and the conversion is applied to the entire file content.
- For more information refer to [5.1.7 Carriage Return, Line Feed and New](#)

Line.

Examples

The following examples use the GET command to receive the contents of a remote file and save it to a local file.

RDML

```
CHANGE FIELD(#JSMCMD) TO('GET TO(order.xml) FROM(/xmldata/orde  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jsmcmd := 'get from(' + #jsmfrom + ') to(' + #jsmto + ')'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SITE

The SITE command allows special FTP server command execution by supplying host-dependent parameters to the remote FTP Server.

```
SITE ----- COMMAND ----- command -----  
-|
```

Keywords

COMMAND The command string to be executed.

Examples

For example, if your remote FTP server is an IBM i you might issue a NAMEFMT command to set the file location inside your remote session.

Setting NAMEFMT to 1 indicates to identify files in all IFS file locations, so all files and paths must be written in IFS notation (e.g. /home/dir).

Setting NAMEFMT to 0 indicates files are to be copied to and from the QSYS.LIB file system using a naming format library/file.member.

RDML

```
CHANGE FIELD(#JSMCMD) TO('SITE COMMAND(NAMEFMT 1)')  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'site command(namefmt 1)'
```

QUOTE

The QUOTE command allows special FTP server command execution by supplying host-dependent commands to the remote FTP Server.

```
QUOTE ----- COMMAND ----- command -----  
-----|
```

Keywords

COMMAND The command string to be executed.

Examples

For example, if your remote FTP server is an IBM i you might issue the RCMD command execute a program or command.

RDML

```
CHANGE FIELD(#JSMCMD) TO('QUOTE COMMAND("RCMD DSPSYSV  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'quote command("RCMD I
```

CREATE

The CREATE command allows a directory to be created on the remote FTP server.

```
CREATE ----- DIR ----- directory path -----|
```

Keywords

DIR Directory path name. This value can be a relative or absolute path.

Examples

RDML

To create a directory named UPLOAD under the root directory:

```
CHANGE FIELD(#JSMCMD) TO('CREATE DIR(/upload)')  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

or to create a directory named UPLOAD under the current working directory:

```
CHANGE FIELD(#JSMCMD) TO('CREATE DIR(upload)')  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

To create a directory named UPLOAD under the root directory:

```
use builtin(jsmx_command) with_args(#jsmhandle 'create dir(/upload) ') to_ge
```

or to create a directory named UPLOAD under the current working directory:

```
use builtin(jsmx_command) with_args(#jsmhandle 'create dir(upload) ') to_get
```

DELETE

The DELETE command allows a directory or file to be deleted from the remote FTP server.

```
DELETE ----- DIR ----- directory path ----->  
>-- FILE ----- file path -----|
```

Keywords

DIR A directory path name is mandatory if no FILE keyword is supplied.
This value can be a relative or absolute path.

FILE A file path name is mandatory if no DIR keyword is supplied.
This value can be a relative or absolute path.

Examples

RDML

To delete a directory and any files under the directory:

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('DELETE DIR(/upload)') T
```

or to delete a single file:

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('DELETE FILE(/upload/or
```

RDMLX

To delete a directory and any files under the directory:

```
use builtin(jsmx_command) with_args(#jsmhandle 'delete dir(/upload) ') to_ge
```

or to delete a single file:

```
use builtin(jsmx_command) with_args(#jsmhandle 'delete file(/upload/order.xr
```

RENAME

The RENAME command allows a file on the remote FTP server to be renamed.

```
RENAME ----- FROM ----- file path ----->
      >-- TO ----- file path -----|
```

Keywords

FROM Existing remote path name.

TO New remote path name.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('RENAME FROM(order.xl
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'rename from(order.xml) to
```

LIST

The LIST command gets a list of file names from the remote FTP server.

```
LIST ---- PATH ----- directory path ----->

>-- FORMAT ----- *AUTO ----->
    *NAME
    *UNIX
    *UNIX8
    *UNIX9
    *ISERIES
    *WINDOWS

>-- DATEFORMAT ---- *NONE ----->
    *DMY
    *MDY
    *YMD
    *YDM

>-- DATALINK ----- *PASV -----|
    *PORT
```

Keywords

- | | |
|--------|--|
| PATH | If no PATH keyword is present, the current directory is used. The path can contain operating system wild characters. |
| FORMAT | The default value *AUTO uses auto-detect list format: <ul style="list-style-type: none">• *AUTO - Auto-detect LIST format.• *NAME - Uses NLST subcommand. 1 column.• *UNIX -List is in UNIX format. 9 columns.• *UNIX8 - List is in UNIX format. 8 columns.• *UNIX9 - List is in UNIX format. 9 columns.• *ISERIES - List is in IBM i format.• *WINDOWS - List is in Windows format. |

When using FTP between IBM i servers it is important to note the FTP attributes LISTFMT value (use CHGFTPA). For example, if the FTP Attributes has LISTFMT(*UNIX) then using LIST PATH(/TMP/*.*)
FORMAT(*ISERIES) will not return any data. Changing to FORMAT(*UNIX) or using the SITE
COMMAND(LISTFMT 0) will return data.

DATEFORMAT The default value *NONE means that no formatting is applied to the date field. Dates are formatted to ISO8601 format.

- *NONE - No date formatting.
- *DMY - Except the date to be in day, month, year order.
- *MDY - Except the date to be in month, day, year order.
- *YMD - Except the date to be in year, month, day order.
- *YDM - Except the date to be in year, day, month order.

DATALINK The DATALINK keyword determines how the "control" session is initiated. The default value is *PASV.

In passive mode, *PASV, the FTP the client initiates both connections to the server. Passive mode causes the FTPService to send a PASV subcommand to the remote FTP server, the FTP server returns the port number of the socket server it has started and a data transfer channel is established from the local server using this port number. This is known as "Passive FTP" and is often used to get around firewall difficulties. Using "Passive" FTP, the local server establishes the second connection, not the remote FTP server. Most firewalls will allow your computer to transfer files in this second connection to the remote FTP server (just as it allowed your computer to establish the first connection).

Port mode, *PORT, causes the FTPService to start a socket server and informs the remote FTP server via the PORT subcommand of the port number this socket server is listening on, the remote FTP server then connects to this socket server and a data transfer channel is established.

Refer to [Technical Specifications](#) for more information.

Comments / Warnings

RDML Clients require the list definition to be passed using the SERVICE_LIST keyword. The list object argument requires one, two or five fields.

If a one field list is being used, then the unprocessed lines are added to the list.

If a two field list is being used, the first field receives the file type and second field receives the file name.

If a five field list is being used, the first field receives the file type , the second field receives the date, the third field receive the time, the fourth field receives the size and fifth field receives the file name.

The type field can contain the following values:

- F - File
- D - Directory
- L - Link

Examples

RDML

```
DEFINE FIELD(#TYPE) TYPE(*CHAR) LENGTH(1)
DEFINE FIELD(#FILE) TYPE(*CHAR) LENGTH(80)
DEF_LIST NAME(#WRKLST) FIELDS((#TYPE) (#FILE)) TYPE(*WORKI

USE BUILTIN(JSM_COMMAND) WITH_ARGS('LIST SERVICE_LIST(TY
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 'list path(/tmp/*.*') formã
```

QUIT

The QUIT command logs out from the remote FTP server and closes the current connection.

```
QUIT ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('QUIT') TO_GET(#JSMST
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 'quit') TO_GET(#jsmxsts #
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSA Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
                *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE

- *SERVICE - read service trace file.
- *TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.message type (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html")

will be returned to the application in this case):

```
com.acme.property.messageType=html
```

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service and disconnects any open system.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 service_unload) to_get(#jsi
```

5.2.4 FTPService Examples

Examples are supplied for the most common client programming languages:

[RDML](#)

[RDMLX](#)

[RPG](#)

5.3 SFTPService

Service Name: **SFTPService**

The SFTPService allows file transfer using SSH2 FTP protocol.

This service uses the open source JCraft SSH2 classes.

Refer to [JCraft](#)

The SFTPService supports the following commands:

5.3.1 [SERVICE_LOAD](#)

5.3.2 [SERVICE_GET](#)

5.3.3 [CONNECT](#)

5.3.4 [LOGIN](#)

5.3.5 [CHGDIR](#)

5.3.6 [GETDIR](#)

5.3.7 [PUT](#)

5.3.8 [GET](#)

5.3.9 [CHMOD](#)

5.3.10 [DELETE](#)

5.3.11 [CREATE](#)

5.3.12 [RENAME](#)

5.3.13 [LIST](#)

5.3.14 [QUIT](#)

5.3.15 [SERVICE_UNLOAD](#)

5.3.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD			
--------------	--	--	--

5.3.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.3.3 CONNECT

The CONNECT command is used to connect to the remote SSH2 FTP server.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CONNECT	HOST	host:port	Mandatory. Remote host. Default port. 22.
	ENCODING		Optional. See ENCODING . Default. *DEFAULT Remote server path encoding.
	COMPRESSION	*YES	Optional. Enable communication compression.
		*NO	Default.
	WAIT	value	Read timeout. The default value is 0.
	TIMEOUT	value	Connection timeout. The default value is 0.

5.3.4 LOGIN

The LOGIN command is used to login to the remote SFTP server.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
LOGIN	USER	value	Mandatory. User profile.
	PASSWORD	value	Optional. User password or private key file password.
	KEY	value	Optional. Public key authentication. The specified value is the PEM encoded SSH private key file. The private key file should have a .pem file extension. The associated PEM encoded SECSH or SSH public key file should have a .pub file extension. For example: remotecompany.pem and remotecompany.pub.

5.3.5 CHGDIR

The CHGDIR command changes the host's working directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CHGDIR	PATH	value	Mandatory. Remote directory.

5.3.6 GETDIR

The GETDIR command returns the host's working directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

GETDIR			
--------	--	--	--

5.3.7 PUT

The PUT command is used to copy a local file to the remote SFTP server.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
PUT	FROM	value	Mandatory. Local file path.
	TO	value	Mandatory. Remote file path.

5.3.8 GET

The GET command is used to copy a remote SFTP server file to a local file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	FROM	value	Mandatory. Remote File path.
	TO	value	Mandatory. Local File path.

5.3.9 CHMOD

The CHMOD command is used to change the permission of a directory or file on the SFTP server.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CHMOD	PATH	value	Mandatory. File path.
	PERMISSION	value	Unix style permission of rwx for owner, group and other. Use a dash for no value. For example: rwxrwxrwx or rw-rw----

5.3.10 DELETE

The DELETE command is used to delete a directory or a file from the SFTP server.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
DELETE	DIR	value	Conditional. Directory path.
	FILE	value	Conditional. File path.

5.3.11 CREATE

The CREATE command is used to create a directory on the SFTP server.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CREATE	DIR	value	Mandatory. Directory path.
	PERMISSIONS	value	Optional. Linux style permissions. For example. rwxr-x---

5.3.12 RENAME

The RENAME command is used to rename a file on the SFTP server.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RENAME	FROM	value	Mandatory. File path.
	TO	value	Mandatory. File path.

5.3.13 LIST

The LIST command is used to retrieve a list of files and directories from the SFTP server.

RDML Clients require the list definition to be passed using the SERVICE_LIST keyword.

The path can contain operating system wild characters.

The list object argument requires one, two or five fields.

If a one field list is being used, then the unprocessed lines are added to the list.

If a two field list is being used, the first field receives the file type and second field receives the file name.

If a five field list is being used, the first field receives the file type, the second field receives the date, the third field receives the time, the fourth field receives the size and fifth field receives the file name.

The type field can contain the following values:

- F - File
- D - Directory
- L - Link

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
LIST	PATH	value	Optional. Directory path. Default. Current directory. The path can contain operating system wild characters.
	TIMEZONE	value	Optional. Timezone of FTP server. Default. JVM default timezone.

5.3.14 QUIT

The QUIT command logs out from the remote SFTP server and closes the current connection.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

QUIT			
------	--	--	--

5.3.15 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.4 HTTPService

Service Name: **HTTPService**

The HTTPService provides a number of different content handlers to receive and send content using the HTTP protocol as follows:

InboundSeparatedValue

InboundXML

InboundXMLBind

InboundJSONBind

InboundNameValue

InboundFile

InboundLocation

InboundStatus

InboundMultiPart

InboundTextHandler

OutboundNameValue

OutboundXML

OutboundXMLBind

OutboundJSONBind

OutboundSeparatedValue

OutboundMultiPart

OutboundFile

The HTTPService supports the following commands:

5.4.1 SERVICE_LOAD

5.4.2 SERVICE_GET

5.4.3 IS

5.4.4 GET

5.4.5 SET

5.4.6 SEND

5.4.7 RECEIVE

5.4.8 BIND

5.4.9 WRITE

5.4.11 SERVICE_UNLOAD

5.4.1 SERVICE_LOAD

If the HTTPService is being used for inbound services, then the SERVICE_LOAD keywords will contain a selection of the HTTP protocol properties sent by the remote HTTP client program.

For a complete list of HTTP protocol property names refer to the [SERVICE_LOAD](#) command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	BIND		Optional. See BIND .
	ARCHIVE		Optional. See ARCHIVE .
	ENCODING		Optional. See ENCODING .
		*BINARY	Archive content with no encoding changes.
	SCHEMA		Optional. See SCHEMA .
	VALIDATING		Optional. See VALIDATING .
	DOMSET		Optional. See DOMSET .
	DOMSETMODEL		Optional. See DOMSETMODEL .
	DOMSETRESULT		Optional. See DOMSETRESULT .

If the ARCHIVE keyword is used the service will use the ENCODING keyword value if it is present or determine the encoding from the protocol content-type entry or use a default value to convert the received byte content to a Unicode string ready for the archiving stage.

The ENCODING keyword is only used when the service is going to archive the received content. The keyword value of *BINARY instructs the service to write

the byte content unaltered to the archive file.

The ARCENCODE keyword controls the archive file content encoding.

If the content type is **not** of type text or XML then the byte content is written to the file unaltered. So posted images and PDF document and other non-text type content will be archive unaltered.

Using the keyword ENCODING (*BINARY) with the ARCHIVE keyword causes all received content to be archived unaltered.

5.4.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.4.3 IS

The IS command allows the function to easily test the type of content received. The IS command uses the CONTENT keyword value and matches it against an internal list of content types and the content type of the received data. If the content matches the returned status is OK else it is NO.

The IS command can be used on received inbound properties and also the received response properties from an outbound content handler.

Syntax:

Command	Keyword	Value	Description
IS	CONTENT	*SV	Comma or tab separated values
		*CSV	Comma separated values
			text/csv
			text/comma-separated-values
*TSV	Tab separated values		
	text/tsv		
	text/tab-separated-values		
	text/x-tab-separated-values		
*XML	XML	application/comma-separated values	
		application/tab-separated values	
		text/xml	
			application/xml
			+xml extension

TEXT	Text	text/ All xml application/json application/edi-x12 application/edifact All separated values
*JSON	JSON	application/json
*FORM	Form POST	application/x- www-form- urlencoded
*MULTIPART	Form upload	multipart/form-data
IMAGE	Image	image/
*GIF	GIF image	image/gif
*PNG	PNG image	image/png
*JPEG	JPEG image	image/jpeg image/jpg
*MPEG	MPEG audio	audio/mpeg
*MPEG4	MPEG4 video	audio/mpeg4 video/mpeg4
*PDF	PDF document	application/pdf
*EXCEL	Excel document	application/vnd.ms- excel
*EDI	X12 or EDIFACT	
*X12		application/edi-x12

*EDIFACT

application/edifact

5.4.4 GET

The GET command allows the program to access HTTP protocol properties. The GET command can be used on received inbound properties and also the received response properties from an outbound content handler.

All properties in the HTTP response from the remote host are available for GET operation when the SEND of the outbound handlers is successful. Two additional properties STATUS and STATUS-MESSAGE are included to allow access to the HTTP response status and message.

For a list of some common HTTP protocol property names see [SERVICE_LOAD](#) command.

The LIST, FRAGMENT and INSTRUCTION keywords are only available if the InboundXMLBind handler is being used.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	PROPERTY	value	HTTP protocol property.
		*LIST	Add all properties to list argument.
	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	INSTRUCTION	value	Conditional. Get XML processing instruction.

5.4.5 SET

The SET command allows the program to include HTTP head properties in the HTTP request or response protocol.

The HTTP property name value pairs are passed as entries in a two-column working list.

The SET command can be called anytime before the last SEND command that creates the HTTP request or response.

The LIST and FRAGMENT keywords are only available if the InboundXMLBind, OutboundXMLBind, InboundJSONBind or OutboundJSONBind handlers are being used.

The INSTRUCTION and DATA keywords are only available if the InboundXMLBind and OutboundXMLBind handlers are being used.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	PROPERTY	*LIST	Set HTTP head properties.
	LIST	Value	Conditional. XML or JSON Bind list.
	FRAGMENT	Value	Conditional. XML or JSON Bind fragment.
	INSTRUCTION	Value	Conditional. XML processing instruction.
	DATA	Value	Conditional. XML processing instruction data.

Example

```
DEFINE FIELD(#PNME) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#PVAL) TYPE(*CHAR) LENGTH(20)
DEF_LIST NAME(#PLST) FIELDS((#PNME) (#PVAL)) TYPE(*WORKIN
```

CHANGE FIELD(#PNME) TO("X-Invoice")

CHANGE FIELD(#PVAL) TO(T9300)

ADD_ENTRY TO_LIST(#PLST)

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET PROPERTY(*LI

5.4.6 SEND

The SEND command allows the program to process available content using the specified content handler. Once the content handler is loaded, this SEND command is passed onto the loaded content handler.

SEND HANDLER

5.4.7 RECEIVE

The RECEIVE command allows the program to process the available content using the specified content handler. Once the content handler is loaded, this RECEIVE command is passed onto the loaded content handler.

RECEIVE HANDLER

5.4.8 BIND

The BIND command is only used with the InboundXMLBind, OutboundXMLBind, InboundJSONBind and OutboundJSONBind handlers.

The BIND command binds the specified service code to the loaded inbound document or creates a new empty outbound document.

The SERVICE keyword value is used in a resource property lookup using the property key service.'value' and service.archive.'value' to locate the binding class and the binding jar file that contains the binding class and supporting classes generated by the XMLBinding or JSONBinding Wizard. If the service binding archive entry does not exist then the binding jar file must exist in the JSM jar subdirectory instead of a user specified directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
BIND	SERVICE	Value	Service name.
	TYPE	*INBOUND	Conditional. Inbound binding.
		*OUTBOUND	Conditional. Outbound binding.
	CONTENT	*JSON	Conditional. Outbound binding type.
		*XML	Default.
	BINDTRACE	*YES	Optional. Trace the inbound bind result.
		*NO	Default.
FILTER		Optional. See FILTER .	

5.4.9 WRITE

The WRITE command is only used with the InboundXMLBind, OutboundXMLBind, InboundJSONBind and OutboundJSONBind handlers. The WRITE command serializes the document object in preparation for the SEND command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
WRITE	INDENT	*YES	Optional. Indent XML content.
		*NO	Default.
	INDENT-AMOUNT	value	Optional. Indent XML amount. Default. 0. If the amount is not zero then indent is enabled.
	DOCTYPE	value	Optional. Include XML <!DOCTYPE> element.
	PUBLIC	value	Conditional. The optional public component of the XML DOCTYPE declaration.
	OMIT-DECLARATION	*YES	Optional. Omit the XML declaration.
		*NO	Default.
	BINDTRACE	*YES	Optional. Trace the outbound bind result.
		*NO	Default.
	FILTER		Optional. See FILTER .

5.4.10 Content Handlers

Inbound content handlers are used to receive incoming data being posted by a HTTP client program and to send a reply.

Outbound content handlers are used to send out going content to a remote HTTP server. The content returned by this remote HTTP server is processed using an inbound content handler.

It is possible to mix and match content handlers. For example, receive posted name value pairs using the InboundNameValue handler and reply with XML content using the InboundXML handler.

The content handler name to content handler class mapping is contained in the HTTPService properties file.

Example

```
handler.InboundFile=com.lansa.jsm.service.InboundFileHandler  
handler.OutboundFile=com.lansa.jsm.service.OutboundFileHandler
```

InboundSeparatedValue

The InboundSeparatedValue handler is used to receive posted separated values and to send separated values as a reply.

When the RECEIVE command of this content handler is executed the following steps occur:

If there is no content or list argument, then an OK status is returned.

The handler checks that the content type belongs to the separated values group of content types, if it does not support the content type, then an error is returned to the program. Use the CONTENT keyword to override this check and to allow the handler to receive the data, it is assumed the data is CSV.

The handler determines the encoding to apply to the receive byte stream to convert it to Unicode.

The handler determines if this Unicode data needs to be archived.

The handler reads the separated values into the working list.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	HANDLER		InboundSeparatedValue
	ARCHIVE		Optional. See ARCHIVE .
	SVROW		Optional. See SVROW .
	SVROWLIMIT		Optional. See SVROWLIMIT .
	SVMODE		Optional. See SVMODE .
	SVHEAD		Optional. See SVHEAD .
	ENCODING		Optional. See ENCODING .
	NUMBERFORMAT		Optional. See NUMBERFORMAT .
	CONTENT		Optional. See CONTENT .
SEPARATOR		Optional. See SEPARATOR .	

Example

RECEIVE HANDLER(InboundSeparatedValue) SVHEAD(ORDER) SERVIC

When the SEND command of this content handler is executed the following steps occur:

The handler uses the value of the CONTENT keyword to determine the type of separated values to create:

- If no content value is available, the handler defaults to application/comma-separated-values
- If the content value is *CSV, then content type will be application/comma-separated-values
- If the content value is *TEXTCSV, then content type will be text/comma-separated-values
- If the content value is *TSV, then content type will be application/tab-separated-values
- If the content value is *TEXTTSV, then content type will be text/tab-separated-values
- If the content value is not a valid separated value content type then CSV data is created, but the specified content type is sent.

The optional SVHEAD keyword allows a header record to be inserted at the beginning of the data.

The handler creates Unicode data using all entries in the list, if there is no list argument then zero length content is created.

The handler determines if this Unicode data needs to be archived.

The handler creates a HTTP response to return to the remote client.

The optional UAFIELD and UAPATH keywords are available to inform the remote client program of a recommended path and filename for the content. LANSAs Integrator UserAgent will add these file and path names to the save popup menu on the response panel.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	---

SEND	HANDLER		InboundSeparatedValue
	UAPATH	value	Optional. User agent save response path.
	UAFILE	value	Optional. User agent save response file name.
	UACACHE	value	Optional. Cache-Control max-age. Default.0.
	UADISPOSITION	value	Optional. Content-Disposition filename.
	NUMBERFORMAT		Optional. See NUMBERFORMAT .
	CONTENT	*CSV	Default. Create CSV content.
		*TSV	Create TSV content.
	ENCODING		Optional. See ENCODING
	SVHEAD		Optional. See SVHEAD
	SVEXCLUDE		Optional. See SVEXCLUDE .
	ARCHIVE		Optional. See ARCHIVE .
	CHARSET	*YES	Optional. Include charset attribute.
		*NO	Do not include charset attribute.
		*TEXT	Default. Include charset attribute if content is text.
SEPARATOR	value	Optional. See SEPARATOR .	

Example

```
SEND HANDLER(InboundSeparatedValue) CONTENT(*CSV) UAFILE(ord
```

InboundXML

The InboundXML handler allows XML requests to be received and XML responses to be sent.

More than one RECEIVE and SEND command can be issued to read and create different sections of a large XML document.

When the RECEIVE command of this content handler is executed the following steps occur:

If there is no content, then an OK status is returned.

The handler checks that the content type belongs to the XML group of content types, if it does not support the content type, then an error is returned to the program. Use the CONTENT keyword to override this check and to allow the handler to receive the data, it is assumed the data is XML.

The handler determines the encoding to apply to the receive byte stream to convert it to Unicode.

The handler determines if this Unicode data needs to be archived.

The handler then transforms the XML and binds the data into the program fields and list.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	HANDLER		InboundXML
	XSL		Mandatory. See XSL
	ENCODING		Optional. See ENCODING .
	CONTENT		Optional. See CONTENT .
	ARCHIVE		Optional. See ARCHIVE .
	SCHEMA		Optional. See SCHEMA .

Example

RECEIVE HANDLER(InboundXML) XSL(ORDER) SERVICE_LIST(...)

When the SEND command of this content handler is executed the following steps occur:

The handler using the XSL keyword transforms the program fields and list into the resultant XML. If a FRAGMENT keyword is present this transformation is stored, as this is only a fragment of the final XML to be sent. When no FRAGMENT keyword is present this transformation is treated as the last and the final output XML is constructed and ready to be sent.

The handler determines if this Unicode data needs to be archived.

The handler uses the value of the CONTENT keyword to determine the value of the HTTP protocol content-type.

- If no content value is available, the handler defaults to application/xml.
- If the content value is *XML, then content type will be application/xml.
- If the content value is *TEXTXML, then content type will be text/xml.

The handler creates a HTTP response to return to the remote client.

The optional UAFILE and UAPATH keywords are available to inform the remote client program of a recommended path and filename for the content. LANSAs Integrator UserAgent will add these file and path names to the save popup menu on the response panel.

The service property inbound.xml.default.contenttype can be used to set the default content type.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		InboundXML
	XSL		Mandatory. See XSL .
	FRAGMENT		Optional. See FRAGMENT .
	OUTPUT		Optional. See OUTPUT .
	UAPATH	value	Optional. User agent save response path.
	UAFILE	value	Optional. User agent save response filename.

UACACHE	value	Optional. Cache-Control max-age. Default.0.
UADISPOSITION	value	Optional. Content-Disposition filename.
ARCHIVE		Optional. See ARCHIVE .
ENCODING		Optional. See ENCODING .
CONTENT		Optional. See CONTENT .
CHARSET	*YES	Optional. Include charset attribute.
	*NO	Do not include charset attribute.
	*TEXT	Default. Include charset attribute if content is text.

Example

SEND HANDLER(InboundXML) XSL(...) UAFILE(...) UAPATH(...) SERV

InboundXMLBind

The InboundXMLBind handler allows XML requests to be received and XML responses to be sent.

When the RECEIVE command of this content handler is executed the following steps occur:

If there is no content, then an OK status is returned.

The handler checks that the content type belongs to the XML group of content types, if it does not support the content type, then an error is returned to the program. Use the CONTENT keyword to override this check and to allow the handler to receive the data, it is assumed the data is XML.

The handler parses the XML content into a DOM object ready for the BIND command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	HANDLER		InboundXMLBind
	ENCODING		Optional. See ENCODING .
	CONTENT		Optional. See CONTENT .
	ARCHIVE		Optional. See ARCHIVE .

Example

```
RECEIVE HANDLER(InboundXMLBind)
```

When the SEND command of this content handler is executed the following steps occur:

The handler determines if this Unicode data needs to be archived.

The handler uses the value of the CONTENT keyword to determine the value of the HTTP protocol content-type.

- If no content value is available, the handler defaults to application/xml.
- If the content value is *XML, then content type will be application/xml.
- If the content value is *TEXTXML, then content type will be text/xml.

The handler creates a HTTP response to return to the remote client.

The optional UAFIELD and UAPATH keywords are available to inform the remote client program of a recommended path and filename for the content. LANSA Integrator UserAgent will add these file and path names to the save popup menu on the response panel.

The service property inbound.xml.default.contenttype can be used to set the default content type.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value	
SEND	HANDLER		InboundXMLBind	
	UAPATH	value	Optional. User agent save response path.	
	UAFIELD	value	Optional. User agent save response filename.	
	UACACHE	value	Optional. Cache-Control max-age. Default.0.	
	UADISPOSITION	value	Optional. Content-Disposition filename.	
	ARCHIVE		Optional. See ARCHIVE .	
	ENCODING		Optional. See ENCODING .	
	CONTENT		Optional. See CONTENT .	
	CHARSET	*YES		Optional. Include charset attribute.
		*NO		Do not include charset attribute.
*TEXT			Default. Include charset	

			attribute if content is text.
--	--	--	-------------------------------

Example

SEND HANDLER(InboundXMLBind) UAFILE(...) UAPATH(...)

InboundJSONBind

The InboundJSONBind handler allows JSON requests to be received and JSON responses to be sent.

When the RECEIVE command of this content handler is executed the following steps occur:

If there is no content, then an OK status is returned.

The handler checks that the content type belongs to the JSON group of content types, if it does not support the content type, then an error is returned to the program. Use the CONTENT keyword to override this check and to allow the handler to receive the data, it is assumed the data is JSON.

The handler parses the JSON content into a JSON object ready for the BIND command.

The encoding of the content is assumed to be UTF-8.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	HANDLER		InboundJSONBind
	CONTENT		Optional. See CONTENT .
	ARCHIVE		Optional. See ARCHIVE .

Example

```
RECEIVE HANDLER(InboundJSONBind)
```

When the SEND command of this content handler is executed the following steps occur:

The handler determines if this Unicode data needs to be archived.

The handler creates a HTTP response to return to the remote client.

The content type is application/json and the content encoding is UTF-8.

The optional UAFIELD and UAPATH keywords are available to inform the remote client program of a recommended path and filename for the content. LANSA Integrator UserAgent will add these file and path names to the save popup menu on the response panel.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		InboundJSONBind
	UAPATH	value	Optional. User agent save response path.
	UAFIELD	value	Optional. User agent save response filename.
	UACACHE	value	Optional. Cache-Control max-age. Default.0.
	UADISPOSITION	value	Optional. Content-Disposition filename.
	ARCHIVE		Optional. See ARCHIVE .

Example

SEND HANDLER(InboundJSONBind)

InboundNameValue

The InboundNameValue handler can send and receive name value pair content. Refer to [5.1.6 Web Browser Content](#).

When the RECEIVE command of this content handler is executed the following steps occur:

If there is no content, then an OK status is returned.

The handler checks that the content type is application/x-www-form-urlencoded, if it is not this type then an error is returned. Use the CONTENT keyword to override this check and to allow the handler to receive the data, it is assumed the data is in the correct format.

The handler then decodes the name value pairs and binds the content to the program depending on the BIND keyword action.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	HANDLER		InboundNameValue
	BIND	*FIELD	Optional. Default. Set LANSAs function fields with received name values.
		*LIST	Optional. Put received name values in specified working list.
		*BOTH	Optional. Set function fields, if field does not exist, put name value in specified working list.
	PREFIX	*YES	Optional. Field has a single letter prefix which needs to be ignored. LANSAsWeb posts name pairs with a single data type prefix letter.
*NO		Default. Field does not have	

			a single letter prefix which needs to be ignored.
	ENCODING		Optional. See ENCODING . Used to convert field values.
	NUMBERFORMAT		Optional. See NUMBERFORMAT .
	CONTENT		Optional. See CONTENT .

Example

RECEIVE HANDLER(InboundNameValue) BIND(*FIELD) SERVICE_LIST

When the SEND command of this content handler is executed the following steps occur:

A name value pair record that is URL encoded is created from the working list argument.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/
SEND	HANDLER		InboundNameValue
	UAPATH	value	Optional. User agent s response path.
	UAFILE	value	Optional. User agent s response filename.
	UACACHE	value	Optional. Cache-Contr max-age. Default. 0.
	UADISPOSITION	value	Optional. Content-Disposition filename.

ENCODING		Optional. See ENCOD Used to encode field v
NUMBERFORMAT		Optional. See NUMBERFORMAT .
CONTENT		Optional. See CONTE
CHARSET	*YES	Optional. Include char attribute.
	*NO	Do not include charset attribute.
	*TEXT	Default. Include char attribute if content is t
SPACE	*PLUS	Default. Encode space +.
	*BLANK	Encode spaces to blan
	*PERCENT20	Encode spaces to %20

Example

SEND HANDLER(InboundNameValue) SERVICE_LIST(...)

InboundFile

The InboundFile handler is used to send and receive files.

The HTTP posted content is saved directly to the specified file.

Command	Keyword	Value	Developer notes for Command/Keyword/Value	
RECEIVE	HANDLER		InboundFile	
	TO	value	Mandatory. File path to receive content.	
	APPEND	*YES		Optional. Append content to existing file.
		*NO		Default.

Example

```
RECEIVE HANDLER(InboundFile) TO(/image-upload/photo.jpeg)
```

When the SEND command of this content handler is executed the following steps occur:

Determine the content-type and read the specified file and send the contents as a HTTP response.

If no CONTENT keyword is present then the Content-Type is determined from the Java Activation Framework FileTypeMap. This is a file extension / MIME type association.

If the HTTP server is an IBM IBM i in MIXED output mode and the content type selected starts with 'text/', the encoding is used to convert the file byte content into Unicode and then converted back to bytes using the value of the CGI-EBCDIC-CCSID property.

If the HTTP server is in BINARY output mode or the content type does not start with 'text/' then the file contents are sent unaltered.

Syntax:

--	--	--	--

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		InboundFile
	FILE	value	Mandatory. File path of content.
	UAPATH	value	Optional. User agent save response path.
	UAFILE	value	Optional. User agent save response filename.
	UACACHE	value	Optional. Cache-Control max-age. Default. 0.
	UADISPOSITION	value	Optional. Content-Disposition filename.
	ENCODING		Optional. See ENCODING Used for MIXED mode content encoding.
	CONTENT		Optional. See CONTENT . Content type is obtained from the file type map.
	CHARSET	*YES	Optional. Include charset attribute.
	*NO	Do not include charset attribute.	
	*TEXT	Default. Include charset attribute if content is text.	

Example

SEND HANDLER(InboundFile) FILE(...)

InboundLocation

The InboundLocation handler can only send a reply, so it is used in conjunction with other inbound content handlers.

When the SEND command of this content handler is executed the following steps occur:

If no URL keyword is supplied, an error is returned.

If the URL keyword value string starts and ends with double quotes, then these are removed.

To create a LANSAWEB procfun query string, include the PROCESS and FUNCTION keywords. If these are present, optional PARTITION and LANGUAGE keywords can be used and the partition will default to the current program partition using the `command.getClient().getPartition()` method and the language will be added to the end if available.

The InboundLocation handler returns the following type of HTTP response.

Status: 303 Redirect to location

Location: url

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		InboundLocation
	URL	value	Mandatory. URL location.
	PROCESS	value	Optional. LANSAWeb process.
	FUNCTION	value	Optional. LANSAWeb function.
	PARTITION	value	Optional. LANSAWeb partition. Default is the partition of the JSM LANSA client function.
	LANGUAGE	value	Optional. LANSAWeb language.

Example

```
SEND HANDLER(InboundLocation) URL(/index.html)
SEND HANDLER(InboundLocation) URL(http://www.lansa.com/index.html)
SEND HANDLER(InboundLocation) URL("http://www.lansa.com/cgi-
bin/jsmdirect?order+period(12)" )
SEND HANDLER(InboundLocation) URL(/cgi-
bin/lansaweb) PROCESS(PROC01) FUNCTION(FUNC01)
```

InboundStatus

The InboundStatus handler can only send a reply, so it is used in conjunction with other inbound content handlers.

When the SEND command of this content handler is executed the following steps occur:

The InboundStatus handler returns the following type of HTTP response.

Status: 200 OK

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		InboundStatus
	STATUS	value	Optional. HTTP status number. Default. 200.
	MESSAGE	value	Optional. HTTP status message Default. OK.

Example

```
SEND HANDLER(InboundStatus) MESSAGE(OK - File has been received)
SEND HANDLER(InboundStatus) MESSAGE(ERROR - Invalid transaction)
```

InboundMultiPart

The InboundMultiPart handler is used to send and receive MIME multipart content.

Refer to [5.1.6 Web Browser Content](#).

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	HANDLER		InboundMultiPart
	FILE	value	Conditional. Name of INPUT tag.
	FIELD	value	Conditional. Name of INPUT tag.
	FILENAME	value	Conditional. Name of INPUT tag. #JSMMSG field is set to the uploaded filename.
	ENCODING		Optional. See ENCODING . Used to convert field values.
	NUMBERFORMAT		Optional. See NUMBERFORMAT .
	TO	value	Conditional. File path to receive content.
	APPEND	*YES	
*NO			Default.

A web browser can send content using the multipart/form-data format.

```
<FORM METHOD="POST" ACTION="http://lansa01/cgi-bin/jsmdirect?upload" ENCTYPE="multipart/form-data">
```

```
<INPUT NAME="COMPANY" TYPE="TEXT" SIZE ="30"/>
<INPUT NAME="FILE1" TYPE="FILE" SIZE ="60"/>
<INPUT NAME="FILE2" TYPE="FILE" SIZE ="60"/>
<INPUT NAME="FILE3" TYPE="FILE" SIZE ="60"/>
<INPUT TYPE="SUBMIT" VALUE="Send order"/>
</FORM>
```

If the specified FILE, FIELD or FILENAME component does not exist then a command status of NOT_EXIST is returned.

If the FILE component does exist but has not been used then NOT_EXIST is also returned.

Example

Receive content from a Web browser post.

```
RECEIVE HANDLER(InboundMultiPart) FIELD(COMPANY)
```

```
RECEIVE HANDLER(InboundMultiPart) FIELD(COMPANY) ENCODING(
```

In the following example the file selected as FILE2 is uploaded to the location and new name given in TO parameter.

```
RECEIVE HANDLER(InboundMultiPart) FILE(FILE2) TO(/image-
upload/photo.jpeg)
```

In this example it is the name of the FILE2 (i.e. C:\images\holiday.jpeg) selected in the browser that is passed to the LANSA function (i.e. holiday.jpeg).

```
RECEIVE HANDLER(InboundMultiPart) FILENAME(FILE2)
```

When the SEND command of this content handler is executed the following steps occur:

Use the working list argument to create a MIME MultiPart response.

The name value pair data and the file data are passed using a list argument.

The first field of the list contains the component's content type of FIELD or FILE.

The second field contains the component's name.

The third field contains the component value or the component file path.

An optional fourth field controls how the file contents are MIME encoded. Possible values are 8bit or base64. The default MIME encoding is 8bit. The MIME body part Content-Type is determined by the file extension using the Java Activation Framework.

The field value encoding is determined by the ENCODING keyword.

Example

```
DEFINE FIELD(#TYPE) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#NAME) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#VALUE) TYPE(*CHAR) LENGTH(100)
DEF_LIST NAME(#WRKLIST) FIELDS(#TYPE #NAME #VALUE) TYPE(*)
```

```
CHANGE FIELD(#TYPE) TO(FIELD)
CHANGE FIELD(#NAME) TO(COMPANY)
CHANGE FIELD(#VALUE) TO(ACME)
ADD_ENTRY TO_LIST(#WRKLIST)
```

```
CHANGE FIELD(#TYPE) TO(FILE)
CHANGE FIELD(#NAME) TO(ORDER_FILE)
CHANGE FIELD(#VALUE) TO("/order/month-order.xml")
ADD_ENTRY TO_LIST(#WRKLIST)
```

```
CHANGE FIELD(#JSMCMD) TO('SEND HANDLER(InboundMultiPart) SE
```

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		InboundMultiPart
	UAPATH	value	Optional. User agent save response path.
	UAFILE	value	Optional. User agent save response filename.
	UACACHE	value	Optional. Cache-Control

		max-age. Default. 0.
UADISPOSITION	value	Optional. Content- Disposition filename.
ENCODING		Optional. See ENCODING . Used to encode field values.
NUMBERFORMAT		Optional. See NUMBERFORMAT .
CHARSET	*YES	Optional. Include charset attribute.
	*NO	Do not include charset attribute.
	*TEXT	Default. Include charset attribute if content is text.

Example

SEND HANDLER(InboundMultiPart) SERVICE_LIST(TYPE,NAME,VALU

InboundTextHandler

The InboundText handler is used to receive text content from a HTTP client program and optional send a text response.

When the RECEIVE command of this content handler is executed the following steps occur:

If there is no content, then an OK status is returned.

The handler then determines the encoding to use to convert the binary text into Unicode text and then updates the field specified in the TO keyword with the converted Unicode value.

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	HANDLER		InboundText
	ENCODING		Optional. See ENCODING . Used to convert binary text to Unicode text.
	TO	field	Mandatory. Bind text to specified field.

Example

```
RECEIVE HANDLER(InboundText) TO(MSG)
```

The SEND command allows a simple plain text response to be created.

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		InboundText
	STATUS	value	Optional. HTTP status number. Default. 200.

MESSAGE	value	Optional. HTTP status message Default. OK.
FROM	field	Optional. Send text from specified field.
UACACHE	value	Optional. Cache-Control max-age. Default.0.
UADISPOSITION	value	Optional. Content-Disposition filename.
ENCODING		Optional. See ENCODING . Used to convert Unicode text to binary text.
CONTENT		Optional. See CONTENT .
CHARSET	*YES	Optional. Include charset attribute.
	*NO	Do not include charset attribute.
	*TEXT	Default. Include charset attribute if content is text.

Example

SEND HANDLER(InboundText) FROM(MSG)

OutboundNameValue

The OutboundNameValue handler can post name value pairs to a remote server using the HTTP protocol.

Refer to [5.1.6 Web Browser Content](#).

When the SEND command of this content handler is executed the following steps occur:

If an optional working list is available name value pair data is created. This working list must have at least two fields, which can be of any name and size. Importantly, the first field supplies the name and the second field supplies the value. The name and values are URL encoded.

If no working list is available then zero length content is created.

If the METHOD keyword is GET, then the HTTP GET protocol is used to access the remote host, if name value pair content exists then this is appended to the end of the URL with a '?' separating the URL and the data.

If the METHOD keyword is POST, then the HTTP POST protocol is used to access the remote host, if name value pair content exists then this is included as content.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		OutboundNameValue
	URI	value	Mandatory. Remote URI.
	HOST	value	Mandatory. Remote server
	USER	value	Optional. Basic authentication user.
	PASSWORD	value	Optional. Basic authentication password.
	PROXY	value	Optional. Proxy server.
	PROXYUSER	value	Optional. Basic authentication user.
	PROXYPASSWORD	value	Optional. Basic

		authentication password.
METHOD		Optional. See METHOD .
VERSION		Optional. See VERSION .
SECURE	*YES	Optional. Use HTTPS protocol.
	*NO	Default. Use HTTP protocol.
ENCODING		Optional. See ENCODING Used to encode field value
WAIT	*YES	Optional. Default. Infinite wait for HTTP response.
	*NO	Do not wait for HTTP response.
	value	Wait read time in milliseconds.
TIMEOUT	value	Optional. Default is 0 which means disabled. Host connection timeout in milliseconds.
LOCALE		Optional. See LOCALE .
NUMBERFORMAT		Optional. See NUMBERFORMAT .
SPACE	*PLUS	Default. Encode spaces to sign.
	*BLANK	Encode spaces to blank.
	*PERCENT20	Encode spaces to %20.

Example

SEND HANDLER(OutboundNameValue) URI(/cgi-
bin/program) HOST(...) SERVICE_LIST(...)

OutboundXML

The OutboundXML handler can post XML to a remote server using the HTTP protocol.

The handler using the XSL keyword transforms the program fields and list into the resultant XML. If a FRAGMENT keyword is present this transformation is stored, as this is only a fragment of the final XML to be sent. When no FRAGMENT keyword is present this transformation is treated as the last and the final output XML is constructed and ready to be sent.

No XSL keyword is needed if the METHOD(GET) keyword is used.

When the SEND command of this content handler is executed the following steps occur:

If content is being posted, then the handler determines if this Unicode data needs to be archived.

The handler uses the value of the CONTENT keyword to determine the value of the HTTP protocol content-type.

- If no content value is available, the handler defaults to application/xml
- If the content value is *XML, then content type will be application/xml
- If the content value is *TEXTXML, then content type will be text/xml

The service property outbound.xml.default.contenttype can be used to set the default content type.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		OutboundXML
	XSL		Conditional. See XSL .
	FRAGMENT		Optional. See FRAGMENT .
	OUTPUT		Optional. See OUTPUT .
	URI	value	Mandatory. Remote URI.
	HOST	value	Mandatory. Remote server.
	USER	value	Optional. Basic authentication user.

PASSWORD	value	Optional. Basic authentication password.
PROXY	value	Optional. Proxy server.
PROXYUSER	value	Optional. Basic authentication user.
PROXYPASSWORD	value	Optional. Basic authentication password.
METHOD		Optional. See METHOD .
VERSION		Optional. See VERSION .
SECURE	*YES	Optional. Use HTTPS protocol.
	*NO	Default. Use HTTP protocol.
WAIT	*YES	Optional. Default is 0. Infinite wait for HTTP response.
	*NO	Do not wait for HTTP response.
	value	Wait read time in milliseconds.
TIMEOUT	value	Optional. Default is 0 which means disabled. Host connection timeout in milliseconds.
LOCALE		Optional. See LOCALE .
CONTENT		Optional. See CONTENT .
ENCODING		Optional. See ENCODING .
ARCHIVE		Optional. See ARCHIVE .
ACTION	value	Optional. Include HTTP SOAPAction property

Example

SEND HANDLER(OutboundXML) XSL(..) URI(/cgi-
bin/program) HOST(...) SERVICE_LIST(...)
SEND HANDLER(OutboundXML) METHOD(GET) URI(/cgi-
bin/program) HOST(...)

OutboundXMLBind

The OutboundXMLBind handler can post XML to a remote server using the HTTP protocol.

To create the XML content, use the BIND, SET and WRITE commands before using the SEND command.

When the SEND command of this content handler is executed the following steps occur:

If content is being posted, then the handler determines if this Unicode data needs to be archived.

The handler uses the value of the CONTENT keyword to determine the value of the HTTP protocol content-type.

- If no content value is available, the handler defaults to application/xml
- If the content value is *XML, then content type will be application/xml
- If the content value is *TEXTXML, then content type will be text/xml

The service property outbound.xml.default.contenttype can be used to set the default content type.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		OutboundXMLBind
	URI	value	Mandatory. Remote URI.
	HOST	value	Mandatory. Remote server.
	USER	value	Optional. Basic authentication user.
	PASSWORD	value	Optional. Basic authentication password.
	PROXY	value	Optional. Proxy server.
	PROXYUSER	value	Optional. Basic authentication user.
	PROXYPASSWORD	value	Optional. Basic authentication

		password.
METHOD		Optional. See METHOD .
VERSION		Optional. See VERSION .
SECURE	*YES	Optional. Use HTTPS protocol.
	*NO	Default. Use HTTP protocol.
WAIT	*YES	Optional. Default is 0. Infinite wait for HTTP response.
	*NO	Do not wait for HTTP response.
	value	Wait read time in milliseconds.
TIMEOUT	value	Optional. Default is 0 which means disabled. Host connection timeout in milliseconds.
LOCALE		Optional. See LOCALE .
CONTENT		Optional. See CONTENT .
ENCODING		Optional. See ENCODING .
ARCHIVE		Optional. See ARCHIVE .
ACTION	value	Optional. Include HTTP SOAPAction property

Example

```
SEND HANDLER(OutboundXMLBind) URI(/cgi-bin/program) HOST(...)
SEND HANDLER(OutboundXMLBind) METHOD(GET) URI(/cgi-
bin/program) HOST(...)
```

OutboundJSONBind

The OutboundJSONBind handler can post JSON to a remote server using the HTTP protocol.

To create the JSON content, use the BIND, SET and WRITE commands before using the SEND command.

When the SEND command of this content handler is executed the following steps occur:

If content is being posted, then the handler determines if this Unicode data needs to be archived.

The content type is application/json and the content encoding is UTF-8.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		OutboundJSONBind
	URI	value	Mandatory. Remote URI.
	HOST	value	Mandatory. Remote server.
	USER	value	Optional. Basic authentication user.
	PASSWORD	value	Optional. Basic authentication password.
	PROXY	value	Optional. Proxy server.
	PROXYUSER	value	Optional. Basic authentication user.
	PROXYPASSWORD	value	Optional. Basic authentication password.
	METHOD		Optional. See METHOD .
	VERSION		Optional. See VERSION .
	SECURE	*YES	Optional. Use HTTPS protocol.
		*NO	Default. Use HTTP protocol.

WAIT	*YES	Optional. Default is 0. Infinite wait for HTTP response.
	*NO	Do not wait for HTTP response.
	value	Wait read time in milliseconds.
TIMEOUT	value	Optional. Default is 0 which means disabled. Host connection timeout in milliseconds.
LOCALE		Optional. See LOCALE .
ARCHIVE		Optional. See ARCHIVE .

Example

```
SEND HANDLER(OutboundJSONBind) URI(/cgi-bin/program) HOST(...)
SEND HANDLER(OutboundJSONBind) METHOD(GET) URI(/cgi-
bin/program) HOST(...)
```

OutboundSeparatedValue

The OutboundSeparatedValue handler can send separated values to a remote server using the HTTP protocol.

When the SEND command of this content handler is executed the following steps occur:

If the handler is being used with keyword METHOD(GET) then no working list argument is needed. It is also possible to POST without a working list argument.

The handler uses the value of the CONTENT keyword to determine the type of separated values to create:

- If no content value is available, the handler defaults to application/comma-separated-values
- If the content value is *CSV, then content type will be application/comma-separated-values
- If the content value is *TEXTCSV, then content type will be text/tab-comma-values
- If the content value is *TSV, then content type will be application/tab-separated-values
- If the content value is *TEXTTSV, then content type will be text/tab-separated-values
- If the content value is not a valid separated value content type then CSV data is created, but the specified content type is sent.

The optional SVHEAD keyword allows a header record to be inserted at the beginning of the data.

The handler creates Unicode data using all entries in the list, if there is no list argument then zero length content is created.

The handler determines if this Unicode data needs to be archived.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		OutboundSeparatedValue
	URI	value	Mandatory. Remote URI.
	HOST	value	Mandatory. Remote server.

USER	value	Optional. Basic authentication user.
PASSWORD	value	Optional. Basic authentication password.
PROXY	value	Optional. Proxy server.
PROXYUSER	value	Optional. Basic authentication user.
PROXYPASSWORD	value	Optional. Basic authentication password.
METHOD		Optional. See METHOD .
VERSION		Optional. See VERSION .
SECURE	*YES	Optional. Use HTTPS protocol.
	*NO	Default. Use HTTP protocol.
WAIT	*YES	Optional. Default is 0. Infinite wait for HTTP response.
	*NO	Do not wait for HTTP response.
	value	Wait read time in milliseconds.
TIMEOUT	value	Optional. Default is 0 which means disabled. Host connection timeout in milliseconds.
LOCALE		Optional. See LOCALE .
NUMBERFORMAT		Optional. See NUMBERFORMAT .
CONTENT		Optional. See CONTENT .
ENCODING		Optional. See ENCODING .
SVQUOTE		Optional. See SVQUOTE .
SVHEAD		Optional. See SVHEAD .

ARCHIVE

Optional. See [ARCHIVE](#).

Example

```
SEND HANDLER(OutboundSeparatedValue) URI(/cgi-  
bin/program) HOST(...) SERVICE_LIST(...)  
SEND HANDLER(OutboundSeparatedValue) URI(/cgi-  
bin/program) HOST(...)CONTENT(*TSV) SERVICE_LIST(...)  
SEND HANDLER(OutboundSeparatedValue) URI(/cgi-  
bin/program) HOST(...)CONTENT(text/csv) SERVICE_LIST(...)
```

OutboundMultiPart

The OutboundMultiPart handler can post name value pair data and file data using the multipart/form-data format.

Refer to [5.1.6 Web Browser Content](#).

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		OutboundMultiPart
	URI	value	Mandatory. Remote URI.
	HOST	value	Mandatory. Remote server.
	USER	value	Optional. Basic authentication user.
	PASSWORD	value	Optional. Basic authentication password.
	PROXY	value	Optional. Proxy server.
	PROXYUSER	value	Optional. Basic authentication user.
	PROXYPASSWORD	value	Optional. Basic authentication password.
	VERSION	value	Optional. See VERSION .
	SECURE	*YES	Optional. Use HTTPS protocol.
		*NO	Default. Use HTTP protocol.
	WAIT	*YES	Optional. Default is 0. Infinite wait for HTTP response.
		*NO	Do not wait for HTTP response.
		value	Wait read time in milliseconds.
	TIMEOUT	value	Optional. Default is 0 which means disabled. Host connection timeout in milliseconds.

LOCALE	Optional. See LOCALE .
NUMBERFORMAT	Optional. See NUMBERFORMAT .
ENCODING	Optional. See ENCODING . Used to encode field values.

The name value pair data and the file data are passed using a list argument. The first field of the list contains the component's content type of FIELD or FILE.

The second field contains the component's name.

The third field contains the component value or the component file path.

An optional fourth field controls how the file contents are MIME encoded. Possible values are 8bit or base64. The default MIME encoding is 8bit. The MIME body part Content-Type is determined by the file extension using the Java Activation Framework.

The field value encoding is determined by the ENCODING keyword.

Example

```

DEFINE FIELD(#TYPE) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#NAME) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#VALUE) TYPE(*CHAR) LENGTH(100)
DEF_LIST NAME(#WRKLIST) FIELDS(#TYPE #NAME #VALUE) TYPE(*

CHANGE FIELD(#TYPE) TO(FIELD)
CHANGE FIELD(#NAME) TO(COMPANY)
CHANGE FIELD(#VALUE) TO(ACME)
ADD_ENTRY TO_LIST(#WRKLIST)

CHANGE FIELD(#TYPE) TO(FILE)
CHANGE FIELD(#NAME) TO(ORDER_FILE)
CHANGE FIELD(#VALUE) TO("/order/month-order.xml")
ADD_ENTRY TO_LIST(#WRKLIST)

```

CHANGE FIELD(#JSMCMD) TO("SEND HANDLER(OutboundMultiPart) :

OutboundFile

The OutboundFile handler can send a file to a remote server using the HTTP protocol.

When the SEND command of this content handler is executed the following steps occur:

The handler uses the value of the CONTENT keyword to determine the content type, if no CONTENT keyword is present, then the file extension is used to determine the content type using the MIME file type map.

If an ENCODING keyword is present, then a charset attribute is determined and appended to the content type.

If the value of the METHOD keyword is GET, then no FILE keyword is required as the HTTP GET protocol is used to receive content from the remote HTTP server.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	HANDLER		OutboundFile
	FILE	value	Conditional. File path.
	URI	value	Mandatory. Remote URI.
	HOST	value	Mandatory. Remote server.
	USER	value	Optional. Basic authentication user.
	PASSWORD	value	Optional. Basic authentication password.
	PROXY	value	Optional. Proxy server.
	PROXYUSER	value	Optional. Basic authentication user.
	PROXYPASSWORD	value	Optional. Basic authentication password.
	METHOD		Optional. See METHOD .

VERSION	Optional. See VERSION .
SECURE	*YES Optional. Use HTTPS protocol. *NO Default. Use HTTP protocol.
WAIT	*YES Optional. Default is 0. Wait read time in milliseconds. *NO Do not wait for HTTP response. value Wait read time in milliseconds.
TIMEOUT	value Optional. Default is 0 which means disabled. Host connection timeout in milliseconds.
LOCALE	Optional. See LOCALE .
CONTENT	Optional. See CONTENT .
ENCODING	Optional. See ENCODING . Used to determine charset attribute.

Example

```
SEND HANDLER(OutboundFile) FILE(/upload/order.csv) URI(/cgi-bin/program) HOST(...)
```

5.4.11 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

DO NOT use the SERVICE_UNLOAD command when sending a HTTP response. Use the SEND command to create the response and then use the JSM_CLOSE BIF to close the connection. The byte array response returned to JSM Client program by the SEND command will be become the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.4.12 HTTPService Examples

RDML [HTTP Service \(XML Inbound\)](#)

RDML [HTTP Service \(XML Outbound\)](#)

5.5 HTTPInboundJSONBindService

Service Name: **HTTPInboundJSONBindService**

The HTTPInboundJSONBindService allows Java Script Object Notation content to be received from and sent to a HTTP browser. The JSON Binding Wizard is used to create a bind mapping between program field values and JSON object variables.

The HTTPInboundJSONBindService supports the following commands:

5.5.1 SERVICE_LOAD

5.5.2 SERVICE_GET

5.5.3 BIND

5.11.4 CLOSE

5.5.5 GET

5.5.6 SET

5.5.7 WRITE

5.5.8 SEND

5.5.1 SERVICE_LOAD

On the SERVICE_LOAD event the service stores the HTTP properties for later access by the GET command.

If there is content, the service checks that the content type is application/json, if it does not support the content type, then an error is returned to the program. Use the CONTENT keyword to override this check and to allow the service to receive the data.

The service determines the encoding to apply to the received content to convert it to Unicode.

The JSON Unicode string is converted into a JSON object.

The service determines if this Unicode data needs to be archived.

The service determines if the query string parameters need to be bound to the program fields or list.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	BIND		Optional. See BIND .
	ARCHIVE		Optional. See ARCHIVE .
	CONTENT		Optional. See CONTENT .
	ENCODING		Optional. See ENCODING .

5.5.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.5.3 BIND

The BIND command binds the specified service code to the loaded inbound JSON object or creates a new empty outbound JSON object

The SERVICE keyword value is used in a resource property lookup using the property key service.'value' and service.archive.'value' to locate the binding class and the binding jar file that contains the binding class and supporting classes generated by the JavaScriptBinding Wizard. If the service binding archive entry does not exist then the binding jar file must exist in the JSM jar subdirectory instead of a user specified directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
BIND	SERVICE	value	Service name.
	TYPE	*INBOUND	Conditional. Inbound binding.
		*OUTBOUND	Conditional. Outbound binding.
	BINDTRACE	*YES	Optional. Trace the inbound bind result.
		*NO	Default.
FILTER		Optional. See FILTER .	

5.5.4 CLOSE

The CLOSE command closes the current BIND.

This command is optional and is usually used after the WRITE command.

The SEND command will automatically do a WRITE and CLOSE operation.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

CLOSE			
-------	--	--	--

5.5.5 GET

The GET command is used to read the inbound JSON object.

Fragments use only fields, so the GET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the GET LIST command requires a working list argument.

The GET command also allows the program to access the inbound HTTP protocol properties.

For a complete list of HTTP protocol property names see [SERVICE_LOAD](#) command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	PROPERTY	value	HTTP protocol property.
		*LIST	Add all properties to list argument.

5.5.6 SET

The SET command is used to populate the outbound JSON object.

Fragments use only fields, so the SET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the SET LIST command requires a working list argument.

The SET command can also be used to include HTTP head properties in the HTTP response protocol.

The HTTP property name value pairs are passed as entries in a two-column working list.

The SET command can be called anytime before the SEND command that creates the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	PROPERTY	*LIST	Conditional. Set HTTP head properties.

Example

```
DEFINE FIELD(#PNME) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#PVAL) TYPE(*CHAR) LENGTH(20)
DEF_LIST NAME(#PLST) FIELDS((#PNME) (#PVAL)) TYPE(*WORKIN

CHANGE FIELD(#PNME) TO("X-Invoice")
CHANGE FIELD(#PVAL) TO(T9300)
ADD_ENTRY TO_LIST(#PLST)

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET PROPERTY(*LI
```

5.5.7 WRITE

The WRITE command serializes the document object in preparation for the SEND command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
WRITE	BINDTRACE	*YES	Optional. Trace the outbound bind result.
		*NO	Default.
	FILTER		Optional. See FILTER .

5.5.8 SEND

The SEND command is used to send the serialized JSON string content back as a response to the remote HTTP client program.

To send a custom CORS response, use the SET PROPERTY command to set the HTTP response properties and use the SEND RESPONSE(*EMPTY) to return no content.

The service determines if this Unicode data needs to be archived.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	ARCHIVE		Optional. See ARCHIVE .
	RESPONSE	*CORS	Optional. Send a Cross-Origin Resource Sharing response.
		*EMPTY	Send an empty content response.

5.5.9 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

DO NOT use the SERVICE_UNLOAD command when sending a HTTP response. Use the SEND command to create the response and then use the JSM_CLOSE BIF to close the connection. The byte array response returned to JSM Client program by the SEND command will be become the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.6 HTTPInboundQueryService

Service Name: **HTTPInboundQueryService**

The HTTPInboundQueryService allows an SQL statement to be executed and the result set to be returned to the client program in CSV, HTML or XML format. The service does not need to receive the query information from the client program. A program could set the query internally and only respond to HTTP GET requests.

The HTTPInboundQueryService supports the following commands:

5.6.1 [SERVICE_LOAD](#)

5.6.2 [SERVICE_GET](#)

5.6.3 [GET](#)

5.6.4 [SET](#)

5.6.5 [SEND](#)

5.6.6 [SERVICE_UNLOAD](#)

5.6.1 SERVICE_LOAD

On the SERVICE_LOAD event the service stores the HTTP properties for later access by the GET command.

If there is content, the service checks that the content type is application/x-www-form-urlencoded, if it does not support the content type, then an error is returned to the program. The service determines the encoding to apply to the received content to convert it to Unicode name value pairs and looks for the two reserved names QUERY and CONTENT.

The QUERY name value contains the SQL query to be executed.

The CONTENT name value specifies the format of the SQL result being returned to the client program. The default value for CONTENT is text/html.

- text/csv
- text/html
- text/xml
- application/xml

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	ENCODING		Optional. See ENCODING .

Example client HTML:

```
<HTML>
<BODY>
<FORM METHOD="POST" ACTION="http://lansa01:1099/cgi-
bin/jsmdirect?query">

<TEXTAREA NAME="QUERY" ROWS="10" COLS="50">select * from dc
</BR>
<INPUT NAME="CONTENT" TYPE="TEXT" VALUE="text/html"/>
<INPUT TYPE="SUBMIT" VALUE="Send Query"/>

</FORM>
</BODY>
```

</HTML>

5.6.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.6.3 GET

The GET command also allows the program to access the inbound HTTP protocol properties.

For a complete list of HTTP protocol property names see [SERVICE_LOAD](#) command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

GET	PROPERTY	value	HTTP protocol property.
-----	----------	-------	-------------------------

*LIST Add all properties to list argument.

5.6.4 SET

The SET command is used set the title and caption for a HTML table result. Also the SET command can be used to set the SQL query and the result set content type.

The SET command can also be used to include HTTP head properties in the HTTP response protocol.

The HTTP property name value pairs are passed as entries in a two-column working list.

The SET command can be called anytime before the SEND command that creates the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	TITLE	value	Optional. HTML page title.
		*NONE	Default.
	CAPTION	value	Optional. HTML table caption.
		*NONE	Default.
	HEAD	*NONE	Optional. Show column header.
		*LONG	Use long names.
*SHORT		Default. Use short names.	
QUERY	*LIST	Optional. SQL query. The list argument contains a large SQL statement. An SQL statement is created by concatenating the list entries and replaces the current query.	
	*NONE		
CONTENT	*HTML	Default. text/html.	
	*CSV		

*XML

*TEXTXML

PROPERTY *LIST Conditional. Set HTTP head
properties.

Example

```
DEFINE  FIELD(#PNME) TYPE(*CHAR) LENGTH(20)
DEFINE  FIELD(#PVAL) TYPE(*CHAR) LENGTH(20)
DEF_LIST NAME(#PLST) FIELDS((#PNME) (#PVAL)) TYPE(*WORKIN

CHANGE  FIELD(#PNME) TO("X-Invoice")
CHANGE  FIELD(#PVAL) TO(T9300)
ADD_ENTRY TO_LIST(#PLST)

USE     BUILTIN(JSM_COMMAND) WITH_ARGS('SET PROPERTY(*LI
```

5.6.5 SEND

The SEND command is used to connect to the database, to execute the query and to format the result set.

If a working list is available then the contents of the working list is returned else a database select is done.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value	
SEND	DRIVER	value	Conditional. Database driver name.	
	DATABASE	value	Conditional. Database name.	
	USER	value	Conditional. Database server user login.	
	PASSWORD	value	Conditional. Database server user password.	
	RESULTSET	*UPDATABLE		Optional. ResultSet concurrency mode.
		*READONLY		Default.
	NUMBERFORMAT		Optional. See NUMBERFORMAT .	
	ARCHIVE		Optional. See ARCHIVE .	
	ENCODING		Optional. See ENCODING .	
	UAPATH	value	Optional. User agent save response path.	
UAFILE	value	Optional. User agent save response file.		
UACACHE	value	Optional. Cache-Control max-age. Default. 0.		

UADISPOSITION	value	Optional. Content- Disposition filename.
CHARSET	*YES	Optional. Include charset attribute.
	*NO	Do not include charset attribute.
	*TEXT	Default. Include charset attribute if content is text.

The driver keyword value is a symbolic name used to locate the service's property entry:

```
driver.sqlserver=com.ddtek.jdbc.sqlserver.SQLServerDriver
```

The database keyword value is a symbolic name used to locate the service's property entry:

```
database.northwind=jdbc:datadirect:sqlserver://10.2.1.28:1433
```

Example

```
SEND DRIVER(SQLSERVER) DATABASE(NORTHWIND) USER(user) PA
```

5.6.6 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

DO NOT use the SERVICE_UNLOAD command when sending a HTTP response. Use the SEND command to create the response and then use the JSM_CLOSE BIF to close the connection. The byte array response returned to JSM Client program by the SEND command will be become the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.7 HTTPInboundXMLService

Service Name: **HTTPInboundXMLService**

The HTTPInboundXMLService offers the same capability as a combined HTTPService and InboundXMLHandler content handler.

The HTTPInboundXMLService supports the following commands:

5.7.1 [SERVICE_LOAD](#)

5.7.2 [SERVICE_GET](#)

5.7.3 [GET](#)

5.7.4 [SET](#)

5.7.5 [RECEIVE](#)

5.7.6 [SEND](#)

5.7.7 [SERVICE_UNLOAD](#)

5.7.1 SERVICE_LOAD

On the SERVICE_LOAD event the service stores the HTTP properties for later access by the GET command.

If there is content, the service checks that the content type belongs to the XML group of content types, if it does not support the content type, then an error is returned to the program. Use the CONTENT keyword to override this check and to allow the handler to receive the data, it is assumed the data is XML.

The service determines the encoding to apply to the received content to convert it to Unicode.

The service determines if this Unicode data needs to be archived.

The service determines if schema validation is turned on.

The service determines if the query string parameters need to be bound to the program fields or list.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	BIND		Optional. See BIND .
	ARCHIVE		Optional. See ARCHIVE .
	CONTENT		Optional. See CONTENT .
	ENCODING		Optional. See ENCODING .
	SCHEMA		Optional. See SCHEMA .
	VALIDATING		Optional. See VALIDATING .
	DOMSET		Optional. See DOMSET .
	DOMSETMODEL		Optional. See DOMSETMODEL .
	DOMSETRESULT		Optional. See DOMSETRESULT .
	DOMGET		Optional. See DOMGET .
DOMGETRESULT		Optional. See	

DOMGETRESULT.

5.7.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.7.3 GET

The GET command allows the program to access HTTP protocol properties. For a complete list of HTTP protocol property names see [SERVICE_LOAD](#) command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	PROPERTY	value	HTTP protocol property.
		*LIST	Add all properties to list argument.

5.7.4 SET

The SET command allows the program to include HTTP head properties in the HTTP response protocol.

The HTTP property name value pairs are passed as entries in a two-column working list.

The SET command can be called anytime before the last SEND command that creates the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	PROPERTY	*LIST	Set HTTP head properties.

Example

```
DEFINE FIELD(#PNME) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#PVAL) TYPE(*CHAR) LENGTH(20)
DEF_LIST NAME(#PLST) FIELDS((#PNME) (#PVAL)) TYPE(*WORKIN

CHANGE FIELD(#PNME) TO("X-Invoice")
CHANGE FIELD(#PVAL) TO(T9300)
ADD_ENTRY TO_LIST(#PLST)

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET PROPERTY(*LI
```

5.7.5 RECEIVE

The RECEIVE command is used to process the content with the specified style sheet and receive the data into the program. Multiple receives command can be used each applying different style sheets to extract the required data.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

RECEIVE	XSL		Mandatory. See XSL .
---------	-----	--	--------------------------------------

5.7.6 SEND

The SEND command is used to create XML content to send back as a response to the remote HTTP client program.

When the SEND command of this content handler is executed the following steps occur:

The service using the XSL keyword transforms the program fields and list into the resultant XML. If a FRAGMENT keyword is present this transformation is stored, as this is only a fragment of the final XML to be sent. When no FRAGMENT keyword is present this transformation is treated as the last and the final output XML is constructed and ready to be sent.

The service determines if this Unicode data needs to be archived.

The service uses the value of the CONTENT keyword to determine the value of the HTTP protocol content-type.

- If no content value is available, the handler defaults to application/xml
- If the content value is *XML, then content type will be application/xml
- If the content value is *TEXTXML, then content type will be text/xml

The service creates a HTTP response to return to the remote client.

The optional UAFIELD and UAPATH keywords are available to inform the remote client program of a recommended path and filename for the content. LANSA Integrator UserAgent will add these file and path names to the save popup menu on the response panel.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	XSL		Mandatory. See XSL .
	FRAGMENT		Optional. See FRAGMENT .
	OUTPUT		Optional. See OUTPUT .
	ARCHIVE		Optional. See ARCHIVE .
	CONTENT		Optional. See CONTENT .
	ENCODING		Optional. See ENCODING .
	UAPATH	value	Optional. User agent save response

		path.
UAFILE	value	Optional. User agent save response file.
UACACHE	value	Optional. Cache-Control max-age. Default. 0.
UADISPOSITION	value	Optional. Content-Disposition filename.
CHARSET	*YES	Optional. Include charset attribute.
	*NO	Do not include charset attribute.
	*TEXT	Default. Include charset attribute if content is text.

5.7.7 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

DO NOT use the SERVICE_UNLOAD command when sending a HTTP response. Use the SEND command to create the response and then use the JSM_CLOSE BIF to close the connection. The byte array response returned to JSM Client program by the SEND command will be become the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.8 HTTPInboundXMLBindService

Service Name: **HTTPInboundXMLBindService**

The HTTPInboundXMLBindService allows inbound and outbound bindings created by the XML Binding Wizard to be used instead of style-sheets.

The HTTPInboundXMLBindService supports the following commands:

5.8.1 [SERVICE_LOAD](#)

5.8.2 [SERVICE_GET](#)

5.8.3 [BIND](#)

5.8.4 [CLOSE](#)

5.8.5 [GET](#)

5.8.6 [SET](#)

5.8.7 [WRITE](#)

5.8.8 [SEND](#)

5.8.9 [SERVICE_UNLOAD](#)

5.8.1 SERVICE_LOAD

On the SERVICE_LOAD event the service stores the HTTP properties for later access by the GET command.

If there is content, the service checks that the content type belongs to the XML group of content types, if it does not support the content type, then an error is returned to the program. Use the CONTENT keyword to override this check and to allow the service to receive the data, it is assumed the data is XML.

The service determines the encoding to apply to the received content to convert it to Unicode.

The service determines if this Unicode data needs to be archived.

The service determines if schema validation is turned on.

The service determines if the query string parameters need to be bound to the program fields or list.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	BIND		Optional. See BIND .
	ARCHIVE		Optional. See ARCHIVE .
	CONTENT		Optional. See CONTENT .
	ENCODING		Optional. See ENCODING .
	SCHEMA		Optional. See SCHEMA .
	VALIDATING		Optional. See VALIDATING .
	DOMSET		Optional. See DOMSET .

5.8.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.8.3 BIND

The BIND command binds the specified service code to the loaded inbound document or creates a new empty outbound document.

The SERVICE keyword value is used in a resource property lookup using the property key service.'value' and service.archive.'value' to locate the binding class and the binding jar file that contains the binding class and supporting classes generated by the XMLBinding Wizard. If the service binding archive entry does not exist then the binding jar file must exist in the JSM jar subdirectory instead of a user specified directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
BIND	SERVICE	value	Service name.
	TYPE	*INBOUND	Conditional. Inbound binding.
		*OUTBOUND	Conditional. Outbound binding.
	BINDTRACE	*YES	Optional. Trace the inbound bind result.
		*NO	Default.
FILTER		Optional. See FILTER .	

5.8.4 CLOSE

The CLOSE command closes the current BIND.

This command is usually used after the WRITE command.

This command is optional as the SEND command will close the binding.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

CLOSE			
-------	--	--	--

5.8.5 GET

The GET command is used to read the inbound document object.

Fragments use only fields, so the GET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the GET LIST command requires a working list argument.

The GET command also allows the program to access the inbound HTTP protocol properties.

For a complete list of HTTP protocol property names see [SERVICE_LOAD](#) command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	PROPERTY	value	HTTP protocol property.
		*LIST	Add all properties to list argument.
	INSTRUCTION	value	Optional. Get XML processing instruction.

5.8.6 SET

The SET command is used to populate the outbound document object.

Fragments use only fields, so the SET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the SET LIST command requires a working list argument.

The SET command can also be used to include HTTP head properties in the HTTP response protocol.

The HTTP property name value pairs are passed as entries in a two-column working list.

The SET command can be called anytime before the SEND command that creates the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	PROPERTY	*LIST	Conditional. Set HTTP head properties.
	INSTRUCTION	value	Optional. Set XML processing instruction.
	DATA	value	Conditional. PI data.

Example

```
DEFINE FIELD(#PNME) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#PVAL) TYPE(*CHAR) LENGTH(20)
DEF_LIST NAME(#PLST) FIELDS((#PNME) (#PVAL)) TYPE(*WORKIN

CHANGE FIELD(#PNME) TO("X-Invoice")
CHANGE FIELD(#PVAL) TO(T9300)
ADD_ENTRY TO_LIST(#PLST)
```

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET PROPERTY(*LI

5.8.7 WRITE

The WRITE command serializes the document object in preparation for the SEND command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
WRITE	INDENT	*YES	Optional. Indent XML content.
		*NO	Default.
	INDENT-AMOUNT	value	Optional. Indent XML amount. Default. 0. If the amount is not zero then indent is enabled.
	DOCTYPE	value	Optional. Include XML <code><!DOCTYPE></code> element.
	PUBLIC	value	Conditional. The optional public component of the XML DOCTYPE declaration.
	OMIT-DECLARATION	*YES	Optional. Omit the XML declaration.
		*NO	Default.
	BINDTRACE	*YES	Optional. Trace the outbound bind result.
		*NO	Default.
	FILTER		Optional. See FILTER .

5.8.8 SEND

The SEND command is used to create XML content to send back as a response to the remote HTTP client program.

When the SEND command of this service is executed the following steps occur:

The service determines if this Unicode data needs to be archived.

The service uses the value of the CONTENT keyword to determine the value of the HTTP protocol content-type.

- If no content value is available, the handler defaults to application/xml
- If the content value is *XML, then content type will be application/xml
- If the content value is *TEXTXML, then content type will be text/xml

The service creates a HTTP response to return to the remote client.

The optional UAFILE and UAPATH keywords are available to inform the remote client program of a recommended path and filename for the content. LANSa Integrator UserAgent will add these file and path names to the save popup menu on the response panel.

To send a custom CORS response, use the SET PROPERTY command to set the HTTP response properties and use the SEND RESPONSE(*EMPTY) to return no content.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	ARCHIVE		Optional. See ARCHIVE .
	CONTENT		Optional. See CONTENT .
	ENCODING		Optional. See ENCODING .
	UAPATH	value	Optional. User agent save response path.
	UAFILE	value	Optional. User agent save response file.
	UACACHE	value	Optional. Cache-Control max-age. Default. 0.

UADISPOSITION	value	Optional. Content-Disposition filename.
CHARSET	*YES	Optional. Include charset attribute.
	*NO	Do not include charset attribute.
	*TEXT	Default. Include charset attribute if content is text.
RESPONSE	*CORS	Optional. Send a Cross-Origin Resource Sharing response
	*EMPTY	Send an empty content response.

5.8.9 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

DO NOT use the SERVICE_UNLOAD command when sending a HTTP response. Use the SEND command to create the response and then use the JSM_CLOSE BIF to close the connection. The byte array response returned to JSM Client program by the SEND command will be become the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.9 HTTPInboundSVService

Service Name: **HTTPInboundSVService**

The HTTPInboundSVService offers the same capability as a combined HTTPService and InboundSVHandler content handler, also it allows an invalid data record to be handled and returned to the sender.

The HTTPInboundSVService supports the following commands:

5.9.1 [SERVICE_LOAD](#)

5.9.2 [SERVICE_GET](#)

5.9.3 [GET](#)

5.9.4 [SET](#)

5.9.5 [RECEIVE](#)

5.9.6 [SEND](#)

5.9.7 [SERVICE_UNLOAD](#)

5.9.1 SERVICE_LOAD

On the SERVICE_LOAD event the service stores the HTTP properties for later access by the GET command.

If there is content, the service checks that the content type belongs to the separated value group of content types, if it does not support the content type, then an error is returned to the program. Use the CONTENT keyword to override this check and to allow the service to receive the data.

The service determines the encoding to apply to the received content to convert it to Unicode.

The service determines if this Unicode data needs to be archived.

The service determines if the query string parameters need to be bound to the program fields or list.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	BIND		Optional. See BIND .
	ARCHIVE		Optional. See ARCHIVE .
	CONTENT		Optional. See CONTENT .
	ENCODING		Optional. See ENCODING .

5.9.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.9.3 GET

The GET command allows the program to access HTTP protocol properties and the error row of the bad data record.

To receive the error record, pass a single field working list and each record field is added as an entry.

For a complete list of HTTP protocol property names see [SERVICE_LOAD](#) command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	PROPERTY	value	HTTP protocol property.
		*LIST	Add all properties to list argument.
	ERROR	*ROW	Get error row.
		*RECORD	Get error record.

5.9.4 SET

The SET command allows the program to include HTTP head properties in the HTTP response protocol.

The HTTP property name value pairs are passed as entries in a two-column working list.

The SET command can be called anytime before the last SEND command that creates the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	PROPERTY	*LIST	Set HTTP head properties.

Example

```
DEFINE FIELD(#PNME) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#PVAL) TYPE(*CHAR) LENGTH(20)
DEF_LIST NAME(#PLST) FIELDS((#PNME) (#PVAL)) TYPE(*WORKIN

CHANGE FIELD(#PNME) TO("X-Invoice")
CHANGE FIELD(#PVAL) TO(T9300)
ADD_ENTRY TO_LIST(#PLST)

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET PROPERTY(*LI
```

5.9.5 RECEIVE

When the RECEIVE command of this service is executed the following steps occur:

If there is no content or list argument, then an OK status is returned.

The service determines if this Unicode data needs to be archived.

The service reads the received Unicode content into the list argument.

The optional SVMODE and SVHEAD keywords direct how this Unicode data is added to the list.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	SVROW		Optional. See SVROW .
	SVROWLIMIT		Optional. See SVROWLIMIT .
	SVMODE		Optional. See SVMODE .
	SVHEAD		Optional. See SVHEAD .
	NUMBERFORMAT		Optional. See NUMBERFORMAT .
	CONTENT		Optional. Default. Use HTTP protocol content-type or assume content is CSV.
		*CSV	Read content as CSV data.
		*TSV	Read content as TSV data.
	ARCHIVE		Optional. See ARCHIVE .

5.9.6 SEND

When the SEND command of this service is executed the following steps occur: The service uses the value of the CONTENT keyword to determine the type of separated values to create:

- If no content value is available, the service defaults to content type of the received content. If no protocol content type is found the service defaults to *CSV.
- If the content value is *CSV, then content type will be application/comma-separated-values
- If the content value is *TEXTCSV, then content type will be text/comma-separated-values
- If the content value is *TSV, then content type will be application/tab-separated-values
- If the content value is *TEXTTSV, then content type will be text/tab-separated-values
- If the content value is not a valid separated value content type then CSV data is created, but the specified content type is sent.

The optional SVHEAD keyword allows a header record to be inserted at the beginning of the data.

The service creates Unicode data using all entries in the list, if there is no list argument then zero length content is created.

The service determines if this Unicode data needs to be archived.

The service creates a HTTP response to return to the remote client.

The optional UAFIELD and UAPATH keywords are available to inform the remote client program of a recommended path and filename for the content. LANSA Integrator UserAgent will add these file and path names to the save popup menu on the response panel.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	CONTENT		Optional. Default. Use HTTP protocol content-type.
		*CSV	Create CSV content.

	*TSV	Create TSV content.
ENCODING		Optional. See ENCODING .
SVQUOTE		Optional. See SVQUOTE .
SVHEAD		Optional. See SVHEAD .
SVEXCLUDE		Optional. See SVEXCLUDE .
NUMBERFORMAT		Optional. See NUMBERFORMAT .
ARCHIVE		Optional. See ARCHIVE .
UAPATH	value	Optional. User agent save response path.
UAFILE	value	Optional. User agent save response file.
UACACHE	value	Optional. Cache-Control max-age. Default. 0.
UADISPOSITION	value	Optional. Content-Disposition filename.
CHARSET	*YES	Optional. Include charset attribute.
	*NO	Do not include charset attribute.
	*TEXT	Default. Include charset attribute if content is text.

5.9.7 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

DO NOT use the SERVICE_UNLOAD command when sending a HTTP response. Use the SEND command to create the response and then use the JSM_CLOSE BIF to close the connection. The byte array response returned to JSM Client program by the SEND command will be become the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.10 HTTPOutboundXMLBindService

Service Name: **HTTPOutboundXMLBindService**

The HTTPOutboundXMLBindService allows inbound and outbound bindings created by the XML Binding Wizard to be used instead of style-sheets.

The HTTPOutboundXMLBindService supports the following commands:

5.10.1 SERVICE_LOAD

5.10.2 SERVICE_GET

5.10.3 BIND

5.10.4 CLOSE

5.10.5 GET

5.10.6 SET

5.10.7 WRITE

5.10.8 SEND

5.10.9 SERVICE_UNLOAD

5.10.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	SCHEMA		Optional. See SCHEMA .
	VALIDATING		Optional. See VALIDATING .
	DOMSET		Optional. See DOMSET .

5.10.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.10.3 BIND

The BIND command binds the specified service code to the loaded inbound document or creates a new empty outbound document.

The SERVICE keyword value is used in a resource property lookup using the property key service.'value' and service.archive.'value' to locate the binding class and the binding jar file that contains the binding class and supporting classes generated by the XMLBinding Wizard. If the service binding archive entry does not exist then the binding jar file must exist in the JSM jar subdirectory instead of a user specified directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
BIND	SERVICE	value	Service name.
	TYPE	*INBOUND	Conditional. Inbound binding.
		*OUTBOUND	Conditional. Outbound binding.
	BINDTRACE	*YES	Optional. Trace the inbound bind result.
		*NO	Default.
FILTER		Optional. See FILTER .	

5.10.4 CLOSE

The CLOSE command closes the current BIND.

This command is usually used after the WRITE command.

This command is optional as the SEND command will close the binding.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

CLOSE			
-------	--	--	--

5.10.5 GET

The GET command is used to read the inbound document object.

Fragments use only fields, so the GET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the GET LIST command requires a working list argument.

The GET command also allows the program to access the HTTP response protocol properties.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	PROPERTY	value	HTTP protocol property.
		*LIST	Add all properties to list argument.
	INSTRUCTION	value	Optional. Get XML processing instruction.

5.10.6 SET

The SET command is used to populate the outbound document object.

Fragments use only fields, so the SET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the SET LIST command requires a working list argument.

The SET command can also be used to include HTTP head properties in the HTTP request protocol.

The HTTP property name value pairs are passed as entries in a two-column working list.

The SET command can be called anytime before the SEND command that creates the HTTP request.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	PROPERTY	*LIST	Conditional. Set HTTP head properties.
	INSTRUCTION	value	Optional. Set XML processing instruction.
	DATA	value	Conditional. PI data.

Example

```
DEFINE FIELD(#PNME) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#PVAL) TYPE(*CHAR) LENGTH(20)
DEF_LIST NAME(#PLST) FIELDS((#PNME) (#PVAL)) TYPE(*WORKIN

CHANGE FIELD(#PNME) TO("X-Invoice")
CHANGE FIELD(#PVAL) TO(T9300)
ADD_ENTRY TO_LIST(#PLST)
```

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET PROPERTY(*LI

5.10.7 WRITE

The WRITE command serializes the document object in preparation for the SEND command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
WRITE	INDENT	*YES	Optional. Indent XML content.
		*NO	Default.
	INDENT-AMOUNT	value	Optional. Indent XML amount. Default. 0. If the amount is not zero then indent is enabled.
	DOCTYPE	value	Optional. Include XML <!DOCTYPE> element.
	PUBLIC	value	Conditional. The optional public component of the XML DOCTYPE declaration.
	OMIT-DECLARATION	*YES	Optional. Omit the XML declaration.
		*NO	Default.
	BINDTRACE	*YES	Optional. Trace the outbound bind result.
		*NO	Default.
	FILTER		Optional. See FILTER .

5.10.8 SEND

The SEND command is used to send the XML content to the remote HTTP server.

When the SEND command of this service is executed the following steps occur:

The service determines if this Unicode data needs to be archived.

The service uses the value of the CONTENT keyword to determine the value of the HTTP protocol content-type.

- If no content value is available, the handler defaults to application/xml
- If the content value is *XML, then content type will be application/xml
- If the content value is *TEXTXML, then content type will be text/xml

The service sends a HTTP request to remote server and receives the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	URI	value	Mandatory. Remote URI.
	HOST	value	Mandatory. Remote server.
	USER	value	Optional. Basic authentication user.
	PASSWORD	value	Optional. Basic authentication password.
	PROXY	value	Optional. Proxy server.
	PROXYUSER	value	Optional. Basic authentication user.
	PROXYPASSWORD	value	Optional. Basic authentication password.
	METHOD		Optional. See METHOD .
	VERSION		Optional. See VERSION .

SECURE	*YES	Optional. Use HTTPS protocol.
	*NO	Default. Use HTTP protocol.
WAIT	*YES	Optional. Default is 0. Infinite wait for HTTP response.
	*NO	Do not wait for HTTP response.
	value	Wait read time in milliseconds.
TIMEOUT	value	Optional. Default is 0 which means disabled. Host connection timeout in milliseconds.
LOCALE		Optional. See LOCALE .
CONTENT		Optional. See CONTENT .
ENCODING		Optional. See ENCODING .
ARCHIVE		Optional. See ARCHIVE .

5.10.9 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.11 HTTPOutboundJSONBindService

Service Name: **HTTPOutboundJSONBindService**

The HTTPOutboundJSONBindService allows inbound and outbound bindings created by the JSON Binding Wizard to be used to send and receive JSON content.

The HTTPOutboundXMLBindService supports the following commands:

5.11.1 SERVICE_LOAD

5.11.2 SERVICE_GET

5.11.3 BIND

5.11.4 CLOSE

5.11.5 GET

5.11.6 SET

5.11.7 WRITE

5.11.8 SEND

5.11.9 SERVICE_UNLOAD

5.11.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD			
--------------	--	--	--

5.11.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	Value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.11.3 BIND

The BIND command binds the specified service code to the loaded inbound JSON object or creates a new empty JSON object.

The SERVICE keyword value is used in a resource property lookup using the property key service.'value' and service.archive.'value' to locate the binding class and the binding jar file that contains the binding class and supporting classes generated by the JSONBinding Wizard. If the service binding archive entry does not exist then the binding jar file must exist in the JSM jar subdirectory instead of a user specified directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
BIND	SERVICE	value	Service name.
	TYPE	*INBOUND	Conditional. Inbound binding.
		*OUTBOUND	Conditional. Outbound binding.
	BINDTRACE	*YES	Optional. Trace the inbound bind result.
		*NO	Default.
FILTER		Optional. See FILTER .	

5.11.4 CLOSE

The CLOSE command closes the current BIND.

This command is optional and is usually used after the WRITE command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

CLOSE			
-------	--	--	--

5.11.5 GET

The GET command is used to read the inbound JSON object.

Fragments use only fields, so the GET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the GET LIST command requires a working list argument.

The GET command also allows the program to access the HTTP response protocol properties.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	PROPERTY	value	HTTP protocol property.
		*LIST	Add all properties to list argument.

5.11.6 SET

The SET command is used to populate the outbound JSON object.

Fragments use only fields, so the SET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the SET LIST command requires a working list argument.

The SET command can also be used to include HTTP head properties in the HTTP request protocol.

The HTTP property name value pairs are passed as entries in a two-column working list.

The SET command can be called anytime before the SEND command that creates the HTTP request.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	PROPERTY	*LIST	Conditional. Set HTTP head properties.

Example

```
DEFINE FIELD(#PNME) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#PVAL) TYPE(*CHAR) LENGTH(20)
DEF_LIST NAME(#PLST) FIELDS((#PNME) (#PVAL)) TYPE(*WORKIN

CHANGE FIELD(#PNME) TO("X-Invoice")
CHANGE FIELD(#PVAL) TO(T9300)
ADD_ENTRY TO_LIST(#PLST)

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET PROPERTY(*LI
```

5.11.7 WRITE

The WRITE command serializes the document object in preparation for the SEND command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
WRITE	BINDTRACE	*YES	Optional. Trace the outbound bind result.
		*NO	Default.
	FILTER		Optional. See FILTER .

5.11.8 SEND

The SEND command is used to send the JSON content to the remote HTTP server.

When the SEND command of this service is executed the following steps occur:

The service determines if this Unicode data needs to be archived.

The service sends a HTTP request to remote server and receives the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	URI	value	Mandatory. Remote URI.
	HOST	value	Mandatory. Remote server.
	USER	value	Optional. Basic authentication user.
	PASSWORD	value	Optional. Basic authentication password.
	PROXY	value	Optional. Proxy server.
	PROXYUSER	value	Optional. Basic authentication user.
	PROXYPASSWORD	value	Optional. Basic authentication password.
	METHOD		Optional. See METHOD .
	VERSION		Optional. See VERSION .
	SECURE		*YES Optional. Use HTTPS protocol. *NO Default. Use HTTP protocol.
WAIT		*YES Optional. Default is 0. Infinite wait for HTTP response.	

	*NO	Do not wait for HTTP response.
	value	Wait read time in milliseconds.
TIMEOUT	value	Optional. Default is 0 which means disabled. Host connection timeout in milliseconds.
LOCALE		Optional. See LOCALE .
ARCHIVE		Optional. See ARCHIVE .

5.11.9 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.12 SMTPMailService

The SMTPMailService supports sending email using the Simple Mail Transfer Protocol (SMTP) through an existing mail server. Most email systems that send mail over the internet use the SMTP protocol to send messages from one server to another. SMTP specifies how the two mail systems need to interact and the format of the control messages they exchange to transfer mail.

When generating email messages using the SMTPMailService, applications can specify a wide range of properties such as recipient and sender addresses, subject and body text, content type and character encoding. Applications can add multiple attachments and, optionally, specify that they are to be collected into a single zip archive file to attach to the outgoing message. The service also supports a variety of encryption techniques.

The SMTPMailService uses the Oracle JavaMail API.

Refer to <http://www.oracle.com/technetwork/java/javamail/index.html>.

Related Services

The SMTPMailService is not dependent on other services but is typically used in conjunction with the POP3MailService if messages are also to be received by the application.

The SMTPMailService can be used to create and send an email message to another server, but since the SMTP protocol is limited in its ability to queue messages at the receiving end it is usually used in conjunction with one of two email client protocols, POP3 or IMAP. If the email client is using JSM services to retrieve the email message then the POP3MailService can be used to retrieve any email messages. The POP3MailService facilitates the saving of messages in a server mailbox and periodic downloading of messages from the server.

Technical Specifications

- Typically the domain information (server, ports and maildomain) and from address are defined in the SMTPMailService properties file unless the default values are being used. This ensures a consistent use of the properties and avoids the need to declare this information each time the service is used.
- SMTP is usually configured to operate over TCP port 25.

To support encrypted emails set up the SMTPMailService property with the appropriate recipient and security file details:

```
recipient.certificate.<person>=<public certificate file>
```

- To support signed emails the SMTPMailService properties file must identify the private key store file and public certificate needed by the SMTPMailServer. The following signer properties must be specified for the email recipient:

signer.keystore.<signer>=<private key store file>

signer.certificate.<signer>=<public certificate file>

5.12.1 What can I use the SMTPMailService for?

Email has become an integral part of modern society has many broad ranging applications.

The following examples provide just a few ideas on how and where the SMTPMailService many be applied.

Create and Send a Regular Update to your Sales Team

On completion of the month end reconciliation, which includes updating and summarizing all the sales figures for the month, an email is automatically formulated and sent to regional sales managers. This email can include the summary of the monthly sales and quarterly targets as straight text in the body of the email or as an attachment, such as an excel spreadsheet.

If security is a concern the contents of the email can be encrypted before distribution.

5.12.2 Using the SMTPMailService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application that sends an email using SMTPMailService would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

SET

ADD

SEND

SERVICE_UNLOAD

JSM(X)_CLOSE

Refer to [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.12.3 SMTPMailService Commands

Your application issues commands to the SMTPMailService by passing the command strings through the Java Service Manager using the JSM_COMMAND or JSMX_COMMAND Built-In Function, or an equivalent Built-In Function or API for your chosen development language.

The commands that the SMTPMailService processes are:

SERVICE_LOAD

SET

ADD

SEND

RESET

SERVICE_GET

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

The following service resource properties are read and used to set initial values.

- server
- port
- secure
- mail.domain
- ntlm.domain
- charset
- user
- password
- from.address
- from.name
- session.debug
- simplify

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
---->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

- SERVICE** The name of the service to be loaded - in this case SMTPMailService.
- TRACE** To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.
- The possible values for the TRACE keyword are:
- *NO
 - *YES
 - *ERROR
- The trace option *ERROR will turn on tracing and if the service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.
- TRACE_NAME** This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory.
- Special keyword values are also available for the TRACE_NAME keyword.
- *SERVICE
 - *PROCESS
 - *FUNCTION
 - *JOBNAME
 - *JOBUSER
 - *JOBNUMBER

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(SMTPMAIL  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jmxcommand := 'service_load service(SMTPMailService)'  
use builtin(jmx_command) with_args(#jmxhdl1 #jmxcommand) to_get(#jmxhdl1 #jmxcommand)
```

SET

The SET command is used to prepare the email. In many ways the SET command is an extension to the SEND command. The SET command can be called many times, with one or more keywords to build up the email definition before it is sent. (This is especially useful in RDML where the command length is limited to 256 characters).

In addition the SET command allows a list of email recipients to be nominated as TO, CC or BCC parties. In contrast the SEND command can only nominate a single email address for these keywords.

At least one keyword must be provided to use the SET command. The SET keyword must be used before the SEND keyword.

Optional

SET ----- SERVER ----- host ----->

*>-- SECURE ----- *NO ----->*

**YES*

**IMPLICIT*

**EXPLICIT*

*>-- TO ----- *LIST ----->*

value

*>-- CC ----- *LIST ----->*

value

*>-- BCC ----- *LIST ----->*

value

>-- FROM ----- value ----->

>-- FROM_NAME ----- value ----->

>-- SUBJECT ----- value ----->

>-- USER ----- value ----->

```

>-- PASSWORD ----- value ----->

>-- PORT ----- 25 ----->
      value

>-- MAILDOMAIN ---- domain name ----->

>-- NTLMDOMAIN ---- domain name ----->

>-- CHARSET ----- UTF-8 ----->
      Value

>-- SIMPLIFY ----- *NO ----->
      *YES

>-- HEADER ----- *NONE ----->
      *LIST

>-- CONTENT ----- *NONE ----->
      *PLAIN
      *HTML
      mimetype

>-- BODY ----- value ----->

>-- ENCODING ----- *DEFAULT -----|
      value

```

Keywords

- | | |
|--------|--|
| SERVER | Any server nominated will be used and overrides the server and port information derived from the service properties file. |
| SECURE | Connect to the server using SSL.
Implicit connects to port 465 using SSL.
Explicit connects to port 587 and upgrades to a TLS session. |

- TO** Include a single email address or the *LIST value to nominate any "to" addresses.
- The *LIST value is used if the email is required to be sent to more than one email address. The list of email addresses is passed using the service list. The first field is the address and the optional second field is the display name.
- Alternatively, a single email address can be specified using the TO keyword in the SEND command.
- At least one TO email address must be supplied to SEND an email.
- CC** Include a single email address or the *LIST value to nominate any "copy to" addresses.
- The *LIST value is used if more than one CC email address is required. The list of email addresses is passed using the service list. The first field is the address and the optional second field is the display name.
- Alternatively, a single email address can be specified using the CC keyword in the SEND command.
- BCC** Include a single email address or the *LIST value to nominate any "blind copy to" addresses.
- The *LIST value is used if more than one BCC email address is required. The list of email addresses is passed using the service list. The first field is the address and the optional second field is the display name.
- Alternatively, a single email address can be specified using the BCC keyword in the SEND command.
- FROM** The email address this message originated from.
- The FROM property is typically configured in the SMTPMailService properties file as it usually does not change and this ensures a consistent FROM email address which can be easily changed in just one location if required.
- Alternatively this value may be specified in the SEND command.
- A FROM email address must be supplied to SEND an

email.

FROM_NAME	<p>Indicates a text name to be displayed as the "from" email display name. If a name is not supplied the FROM email address will be displayed.</p> <p>The FROM_NAME property is typically configured in the SMTPMailService properties file as it usually does not change and this ensures a consistent FROM_NAME to match the configured FROM email address.</p>
SUBJECT	<p>Indicates the subject of the email.</p> <p>The *LIST value is used to get the subject text a two column working list. The working list must have two columns and a list entry with the first field with a key value of SUBJECT and the second value field containing the single line of subject text. This mechanism was introduced to allow Unicode subject text, but is now obsolete as the JSM command BIF can now accept Unicode command field.</p> <p>Alternatively this value may be specified in the SEND command.</p>
USER	<p>This is the user profile used to login to the SMTP server. This user name may be case sensitive depending on the server.</p>
PASSWORD	<p>This is the password used to login to the SMTP server. The password may be case sensitive depending on the SMTP server and must match the USER keyword provided.</p>
PORT	<p>Nominate the TCP port to be used for SMTP. The default for SMTP, port 25, is used if an alternative port number is not supplied.</p> <p>If a port other than 25 is used the PORT property is typically configured in the SMTPMailService properties file rather than setting the PORT keyword each time an email is sent.</p>
MAILDOMAIN	<p>This is the mail domain issued by the SMTP HELO subcommand to identify the server initiating the mail</p>

transaction.

The default value used is the localhost domain.

The MAILDOMAIN property is typically configured in the SMTPMailService properties file.

- NTLMDOMAIN** This is the NTLM domain for NTLM authentication.
The NTLMDOMAIN property is typically configured in the SMTPMailService properties file.
- CHARSET** The character set encoding to be applied to body text and the subject.
The default value applied is iso-8859-1.
The CHARSET property is typically configured in the SMTPMailService properties file.
- SIMPLIFY** Simplify a multipart plain text message into a simple plain text message.
- HEADER** This allows MIME header fields to be sent with the message.
The *NONE value clears the all headers.
The *LIST value is used supply a list of name values. The first field is the field and the second field is the value.
- CONTENT** This allows up to three multipart/alternative text messages to be sent.
The *NONE value clears all three text messages.
The *PLAIN value creates the text/plain content.
The *HTML value creates the text/html content.
The MIME type value creates the custom content. ie. text/enriched.
See SEND command on how to create the body text from the working list argument.
- BODY** A file can be nominated to contain the body text. If the specified file does not exist the body text is created from the service list argument if it is provided.
If the BODY keyword is used any reference to a working

list is ignored as the body text is taken from the file and converted to Unicode using the ENCODING keyword value.

ENCODING Used to specify what encoding must be applied to the byte content to convert it to a Unicode string. The default value for the ENCODING keyword is *DEFAULT.
Refer to See [ENCODING](#) for more information.

Comments / Warnings

Any keyword values set up using the SET command will be overridden by use of the equivalent keyword on the SEND command.

Examples

RDML

Sets who the email is from and addresses the email to a single email address.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET FROM(me@compan
```

or

Sets the email to be sent to the group of email addresses included in the list #TOLIST. Issues a second SET command to indicate who the email is from.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET TO(*LIST) SERVI
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET FROM (me@compan
```

RDMLX

Process a group of email addresses separated by a semicolon, into a working list and set this list of emails as to addressees on the email.

* Set the to address(es)

```
#start := 1
```

```
dountil (#pos = 0)
```

```
if (#ccaddress.cursize > #start)
#pos := #toaddress.positionof( ';', #start )
else
#pos := 0
endif
```

```
if (#pos = 0)
#jsmemail := #toaddress.substring( #start )
else
#jsmemail := #toaddress.substring( #start, (#pos - #start) )
#start := #pos + 1
endif
```

```
if (#jsmemail *ne *blank)
add_entry to_list(#tolist)
endif
```

```
enduntil
```

```
use builtin(jsmx_command) with_args(#jsmhandle 'set to(*list) ') to_get(#jsms
execute subroutine(check) with_parms(#jsmsts #jsmmsg)
```

ADD

The ADD command is used to include one or more file attachments on the email being formatted.

Files can be added directly to the email, or the ZIP keyword can be used to create an archive file of a nominated list of files. The archive file is then included as a single attachment on the email.

The ADD keyword must be used before the SEND keyword.

Optional

```
ADD ----- ATTACHMENT ----- *LIST -----  
>  
      value  
  
>-- ZIP ----- value -----|
```

Keywords

ATTACHMENT A single file can be attached to the email by nominating the file path and name in the ATTACHMENT keyword.

To include more than one file attachment, create a working list of files where the first field in the list is the file path and name. The optional second field in the working list is the attachment display name or the zip entry name. Use the *LIST value for the ATTACHMENT keyword and include the working list information as a service list.

If no file attachments are included the ADD command will be ignored.

ZIP The name to be given to the zip archive created and attached to the email. The file name should include the suffix .zip for easy processing when the email is received.

The ZIP keyword will be ignored unless a list of files is provided using *LIST in the ATTACHMENT keyword.

Examples

RDML

Include a single text file as an attachment to the email.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('ADD ATTACHMENT(orc
```

RDMLX

Zip a group of files and attach the single zipped archive to the email.

```
define field(#file name) type(*char) length(255)  
def_list name(#filelist) fields(#file name) type(*Working)
```

```
clr_list named(#filelist)
```

```
#file name := order.xml  
add_entry to_list(#filelist)  
#file name := history/lastorders.pdf  
add_entry to_list(#filelist)
```

```
#jsmcmd := 'add attachment(*list) zip(orderstatus.zip)'
```

```
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SEND

The SEND command sends the email.

You can use the TO, CC, BCC, FROM and SUBJECT keywords in the SEND command to override the values set by the SMTPMailService properties file or the SET command.

If the SEND command has a working list argument the list entries will be used to create the message body. The first list field must contain the text and an optional second field is used to control the concatenation process. If no second field is present, or the field has a value of blank, then the default option value of NA is used.

The possible option values are:

SB - Add a space before appending the text entry, except for the first entry.

NB - Add a new line before appending the text entry, except for the first entry.

SA - Add a space after appending the text entry, except for the last entry.

NA - Add a new line after appending the text entry.

If the second field value is not one of the above then no additional action is taken and the next entry is appended to the previous entries.

Optional

```
SEND ----- TO ----- value ----->
>-- CC ----- value ----->
>-- BCC ----- value ----->
>-- FROM ----- value ----->
>-- SUBJECT ----- value ----->
>-- SIGNER ----- value ----->
>-- MULTIPARTSIGNED --- *NO ----->
>
```

**YES*

```
>-- ENCRYPT ----- none ----->
    *DES
    *DES-EDE3
    *RC2-40
    *RC2-64
    *RC2-128

>-- CONTENT ----- *PLAIN ----->
    *HTML
    value

>-- BODY ----- value ----->

>-- ENCODING ----- *DEFAULT -----|
    value
```

Keywords

TO	Allows the a single email address to be nominated for Alternatively use the SET command to specify a single
CC	Allows a single email address to be nominated for the Alternatively use the SET command to specify a single
BCC	Allows a single email address to be nominated for the Alternatively use the SET command to specify a single
FROM	The email address this message originated from. The FROM property is typically configured in the SM ensures a consistent FROM email address which can b Alternatively this value may be specified in the SET cc A FROM email address must be supplied to SEND an
SUBJECT	Indicates the subject of the email. Alternatively this value may be specified in the SET cc
SIGNER	The signer value is comprised of two parts, the name a

to access the appropriate private key store and security
To send signed emails, a private key and public certificate
identified in the SMTPMailService properties file for t
The X509 public certificate file to be used is obtained
The PKCS8 keystore file to be used is obtained from tl
password component.

For example, if your signer name is SMTPSIGN your

```
signer.keystore.SMTPSIGN=test-input/smtp-signer-k  
signer.certificate.SMTPSIGN=test-input/smtp-signer
```

Then to send a signed email you would issue a JSM co

```
use builtin(jsmx_command) with_args(#jsmxhdl1 's
```

There is no need to send a public certificate to the part
automatically included in the email.

Important: There is no need to send private key store

MULTIPARTSIGNED Allows the service to send a signed message in multipa
*YES - Send message in multipart/signed format.
*NO - Do not send message in multipart/signed format

ENCRYPT Encryption is the process of transforming a clear-text r
meaningless sequence of bits (ciphertext). A cipher is t
ciphertext output and vice versa.

If the email is to be encrypted, an encryption algorithm
an email but the SMTPMailService supports the follow

***DES** - Data Encryption Standard is a block cipher, (i.
length algorithm. It uses 56-bit keys where any 56-bit

***DES-EDE3** Triple DES improves the security of DE
keys. This way the effective key length becomes 168 b

***RC2-40**, ***RC2-64** and ***RC2-128** - are versions of t
respectively. By definition RC2 encryption is a variabl
to 128 bits in 8-bit increments.

Once the encryption algorithm has been determined th

email. All modern ciphers use keys together with plain is supplied to the decryption function to recover plaintext of the cipher (which are usually made public) but in the To send encrypted emails requires the public certificate The person receiving your email must send you their properties file for each email recipient indicating the location For example:

recipient.certificate.john.doe@lansa.com.au=test-inp

CONTENT

The most commonly used email content-types are text/*HTML).

The default content value of *PLAIN indicates a MIME A value of *HTML indicates a MIME type of text/html the email to be formatted using HTML tags.

Alternative content-types can also be assigned using the For example, text/enriched.

BODY

A file can be nominated to contain the body text. If the service list argument if it is provided.

If the BODY keyword is used any reference to a working converted to Unicode using the ENCODING keyword

ENCODING

Used to specify what encoding must be applied to the list the ENCODING keyword is *DEFAULT.

Refer to See [ENCODING](#) for more information.

Examples

RDML

Set up the subject of the email and include the body text of the email in a working list. Additional detail may have been defined using the SET and ADD commands prior to the SEND command, or default values from the SMTPMailService property file will be applied.

```
USE BUILTIN(TCONCAT) WITH_ARGS('SEND SUBJECT(' #SUBJECT ')');  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

Set up the subject of the email and include the body text in a file. Additional detail may have been defined using the SET and ADD commands prior to the SEND command, or default values from the SMTPMailService property file will be applied.

```
#jsmcmd := 'send subject(' + #subject + ') body(emailtext.txt) encrypt(*DES)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

RESET

The RESET command clears the message text, subject, recipient and attachment lists.

```
RESET ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(RESET) TO_GET(#JSMS'
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 reset) TO_GET(#jsmxsts #
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSA Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
                *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE

- *SERVICE - read service trace file.
- *TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.message type (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html")

will be returned to the application in this case):

```
com.acme.property.messageType=html
```

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service and removes the temporary directory.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhde1 service_unload) to_get(#jsi
```

5.12.4 SMTPMailService Examples

Examples are supplied for the most common client programming languages:

[RDML](#)

[RPG](#)

5.12.5 Troubleshooting

If you receive the message "Require SMTP Server":

1. Check the JSM instance you are connecting to is correct and has been successfully started.
2. Verify that the correct SERVER information has been provided in SET / SEND commands, or alternatively in the SMTPMailService properties file.

5.13 SMTPMailAttachmentSignatureService

Service Name: **SMTPMailAttachmentSignatureService**

The SMTPMailAttachmentSignatureService:

- permits applications to send a single email attachment and a digital signature.
- uses the Oracle JavaMail API.

Refer to [JavaMail](#)

5.13.1 SERVICE_LOAD

Loads and initializes the service.

The following service resource properties are read and used to set initial values.

- server
- secure
- port
- mail.domain
- ntlm.domain
- charset
- user
- password
- from.address
- from.name
- session.debug

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD			
--------------	--	--	--

5.13.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.13.3 SET

The SET command allows the current node to be set to the specified path.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	SERVER	value	Optional.
	SECURE	value	Optional.
	PORT	value	Optional.
	MAILDOMAIN	value	Optional.
	NTLMDOMAIN	value	Optional.
	CHARSET	value	Optional.
	USER	value	Optional.
	PASSWORD	value	Optional.
	FROM	value	Optional.
	TO	*LIST	Optional.
		value	Optional.
	CC	*LIST	Optional.
		value	Optional.
	BCC	*LIST	Optional.
		value	Optional.
	SUBJECT	value	Optional.
	ATTACHMENT	value	Optional.
	ATTACHMENTNAME	value	Optional.
	KEYSTORE	value	Optional. keystore:password.

5.13.4 RESET

The RESET command clears the subject, recipients and attachment.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

RESET			
-------	--	--	--

5.13.5 SEND

The SEND command sends the email.

You can use the FROM, TO, CC, BCC, SUBJECT, ATTACHMENT and ATTACHMENTNAME keywords in the SEND command to override the values set by the SMTPMailAttachmentSignatureService properties file or the SET command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	FROM	value	Optional.
	TO	value	Optional.
	CC	value	Optional.
	BCC	value	Optional.
	SUBJECT	value	Optional.
	ATTACHMENT	value	Optional.
	ATTACHMENTNAME	value	Optional.
	SIGNER	value	Optional. keystore entry:password

5.13.6 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.14 POP3MailService

The POP3MailService permits applications to receive email messages using the Post Office Protocol (POP3) from an existing mail server and to process their contents. Most mail servers support the POP3 protocol for mail clients to receive messages.

The POP3MailService uses the Oracle JavaMail API.

Refer to <http://www.oracle.com/technetwork/java/javamail/index.html>.

Related Services

The POP3MailService is not dependent on other services but is often used in conjunction with the SMTPMailService if email messages are also initiated by the application.

The SMTPMailService can be used to create and send an email message to another server, but since the SMTP protocol is limited in its ability to queue message at the receiving end it is usually used in conjunction with one of two email client protocols, POP3 or IMAP. If the email client is using JSM services to retrieve the email message then the POP3MailService can be used to retrieve any email messages. The POP3MailService facilitates the saving of messages in a server mailbox and periodic downloading of messages from the server.

Technical Specifications

- The POP3 protocol works over a TCP/IP connection using TCP, usually on network port 110.
- Typically the server (and often the user and password) is defined in the POP3MailService properties file. These values will be used as the default connection information when the service is used (unless alternative values are provided in the application code). For example:

```
# POP3MailService resources ( Default )  
#  
server=10.2.0.12  
user=user1  
password=xxxxx  
# port=110  
# recipient.keystore.name=pki/person-key.p8  
# recipient.certificate.name=pki/person-cert.cer
```

5.14.1 What can I use the POP3MailService for?

Using this service, applications can retrieve one or more messages from the email store for a nominated user, access the details and contents of the individual messages, including the sender details, subject, body text and file attachments, and request that processed messages be deleted from the mail server.

The POP3MailService supports decrypting secure messages that have been encrypted with the recipient's public key.

5.14.2 Using the POP3MailService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application that reads an email using POP3MailService would typically issue the following sequence of commands:

JSM(X)_OPEN**JSM(X)_COMMANDs**

SERVICE_LOAD

OPEN

[loop]

GET (repeated)

READ or SAVE

DELETE

[end loop]

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

The GET command is typically used repeatedly to retrieve the next message and then get the details of the current message. The SAVE or READ command is used to process any file attachments.

Refer to [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.14.3 POP3MailService Commands

Your application issues commands to the POP3MailService by passing the command strings through the Java Service Manager using the JSM_COMMAND or JSMX_COMMAND Built-In Function, or an equivalent Built-In Function or API for your chosen development language.

The commands that the POP3MailService processes are:

SERVICE_LOAD
OPEN
GET
SAVE
READ
DELETE
CLOSE
SERVICE_GET
SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

The following service resource properties are read and used to set initial values.

- server
- secure
- port
- user
- password
- timezone
- session.debug

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
---->
```

Optional

```
>-- TIMEZONE ----- value ----->
```

```
>-- TRACE ----- *NO ----->
```

```
*YES
```

```
*ERROR
```

```
>-- TRACE_NAME ----- name -----|
```

```
*SERVICE
```

```
*PROCESS
```

```
*FUNCTION
```

```
*JOBNAME
```

```
*JOBUSER
```

```
*JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

- SERVICE** The name of the service to be loaded - in this case POP3MailService.
- TIMEZONE** The TIMEZONE value provided in the POP3MailService properties file can be overridden using the TIMEZONE keyword on the SERVICE_LOAD command.
Refer to [5.1.3 Time Zones](#) for more information.
- TRACE** To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.
The possible values for the TRACE keyword are:
- *NO
 - *YES
 - *ERROR
- The trace option *ERROR will turn on tracing and if the service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.
- TRACE_NAME** This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory.
Special keyword values are also available for the TRACE_NAME keyword.
- *SERVICE
 - *PROCESS
 - *FUNCTION
 - *JOBNAME
 - *JOBUSER
 - *JOBNUMBER

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(POP3MAIL  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jsmcommand := 'service_load service(pop3mailservice)'  
use builtin(jsmx_command) with_args(#jsmxhdl1 #jsmcommand) to_get(#jsn
```

OPEN

The OPEN command will login into the POP3 server and read any email messages stored for that user.

If the server, user and password are not defined in the service's properties file they must be provided in the OPEN command.

Internally a folder object is created to hold these messages. The GET and DELETE commands work against this internal folder object.

Optional

```
OPEN ----- SERVER ----- host ----->

>-- SECURE ----- *NO ----->
      *YES
      *IMPLICIT
      *EXPLICIT

>-- USER ----- value ----->

>-- PASSWORD ----- value ----->

>-- PORT ----- 110 ----->
      number

>-- RECIPIENT ----- name:password -----|
```

Keywords

- | | |
|--------|--|
| SERVER | The server address for the POP3 server.
The server address must be provided if it has not been configured.
The SERVER property is typically configured in the POP3MailService properties file. |
| SECURE | Connect to the server using SSL.
Implicit connects to port 995 using SSL.
Explicit connects to port 110 and upgrades to a TLS session. |
| USER | The POP3 mail user's login identification. |

The mail user must be provided if it has not been configured in the JSM command. The USER property is typically configured in the POP3MailServer component.

PASSWORD The password corresponding to the POP3 User provided as the USER property. The mail user's password must be provided if it has not been configured in the JSM command. The PASSWORD property is typically configured in the POP3MailServer component.

PORT Nominate the TCP port to be used for POP3. The default for POP3, port 110, is used if an alternative port number is not provided. If a port other than 110 is used the PORT property is typically configured in the JSM command.

RECIPIENT This keyword is required to decrypt secure messages that have been encrypted using the POP3SIGN keyword. The recipient value is comprised of two parts, the name and password. The name is used to access the appropriate private key store and security certificate. The X509 public certificate file to be used is obtained from the JSM command. The PKCS8 keystore file to be used is obtained from the property component. So for example if your recipient name is POP3SIGN your POP3 command would be like:

```
recipient.keystore.POP3SIGN=test-input/pop3-recipient-key.p
recipient.certificate.POP3SIGN=test-input/pop3-recipient-cert
```

Then to receive an encrypted email you would issue a JSM command like:

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'open recipient')
```

Examples

RDML

In this example the SERVER, USER and PASSWORD keyword values are explicitly defined as part of the command.

```
USE BUILTIN(TCONCAT) WITH_ARGS('OPEN SERVER(' #JMSERVER
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #JSMPSWRD ')') TC
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

In this example the SERVER, USER and PASSWORD keyword values are explicitly defined as part of the command.

```
#jsmcmd := 'open server(' + #jsmserver + ') user(' + #jsmuser + ') password(' +  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

GET

The GET command retrieves email details from the mail server. The GET command is usually used repetitively to get all the messages and appropriate information pertaining to the messages.

Conditional

```
GET ----- OBJECT ----- none -----|
      *MESSAGECOUNT
      *FIRSTMESSAGE
      *NEXTMESSAGE
      *TOADDRESS
      *CCADDRESS
      *FROMADDRESS
      *SUBJECT
      *SENTDATE
      *ATTACHMENTS
      *UID
      *HEADER
```

Keywords

OBJECT This keyword must be specified on the GET command with one of the predetermined values.

A value of *MESSAGECOUNT will return the number of messages in the folder in the JSM command message parameter.

A value of *FIRSTMESSAGE sets the current message to the first message in the folder. If no message is available the command status field is set to NOMAIL.

Specifying a value of *NEXTMESSAGE will get the next mail message in the folder and set this as the current message. If there are no more messages the command status field is set to NOMAIL. If a message is found the command status is returned as OK. The *NEXTMESSAGE processing is usually followed by another GET command to retrieve the details of the message e.g. the subject, from address or attachments, or the SAVE command to get the actual body of the message.

A value of *TOADDRESS will return the email address of the email recipient(s) into the JSM command message parameter or, if a list is provided, into the working list.

A value of *CCADDRESS will return the email address of the email recipient(s) into the JSM command message parameter or, if a list is provided, into the working list.

A value of *FROMADDRESS will return the email address of the email sender into the JSM command message parameter or, if a list is provided, into the working list.

A value of *FROMADDRESS will return the email address of the email sender into the JSM command message parameter or, if a list is provided, into the working list.

A value of *SUBJECT will return the subject of the current message in the JSM command message parameter.

A value of *SENTDATE will return the date the email was sent in the JSM command message parameter.

Specifying *ATTACHMENTS will return a working list with an entry detailing the file name of each attachment on the current email.

A value of *UID will return the unique identifier of the current message.

Specifying *HEADER and using the keyword FIELD to specify which MIME header field to be returned.

Examples

The following examples use the GET command to retrieve email details from the mail server.

RDML

```
* loop through all the messages
```

```
BEGIN_LOOP
```

```
* Get message
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET OBJECT(*NEXTME
```

```
IF COND('#JSMSTS *EQ NOMAIL')
```

```
LEAVE
```

```
ENDIF
```

```
* Get subject
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET OBJECT(*SUBJECT)
CHANGE FIELD(#SUBJECT) TO(#JSMMSG)
```

```
* Get from addresses
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET OBJECT(*FROMAI
CHANGE FIELD(#FROM) TO(#JSMMSG)
```

```
* <<do something with email details>>
```

```
END_LOOP
```

RDMLX

```
* loop through all the messages
begin_loop
```

```
* Get the next message
use builtin(jsmx_command) with_args(#jsmhandle 'get object(*nextmessage)')
if (#jsmsts = NOMAIL)
leave
endif
```

```
* Get subject
use builtin(jsmx_command) with_args(#jsmhandle 'get object(*subject)') to_get
#jsmsubject := #jsmmsg
```

```
* Get from address
use builtin(jsmx_command) with_args(#jsmhandle 'get object(*fromaddress)')
#jsmfrom := #jsmmsg
```

```
* <<do something with email details>>
```

```
end_loop
```

SAVE

The SAVE command is used to save various components of the current message to a local file (or files).

Required

```
SAVE ----- OBJECT ----- *TEXT ----->  
*ATTACHMENT  
*ATTACHMENTS
```

Conditional

```
>-- FILE ----- value ----->
```

```
>-- DIR ----- value ----->
```

Optional

```
>-- ENTRY ----- 1 ----->  
value
```

```
>-- SELECT ----- value -----|
```

Keywords

OBJECT This keyword specifies what part of the current message is to be saved.

A value of *TEXT indicates the body text of the message is to be saved. If the message content-type does not start with text/ or multipart/ the command returns a value of NOTEXT in the status field.

Use a value of *ATTACHMENT to nominate a specific attachment to be saved. Use this value in combination with the ENTRY or SELECT keywords to identify the attachment to be saved. If neither ENTRY nor SELECT is specified the first attachment is saved. If there are no attachments associated with the current message the command status is returned as

NOATTACHMENT.

Use a value of *ATTACHMENTS to save all attachments to the directory specified by the keyword DIR using the attachment file name. If there are no attachments associated with the current message the command status is returned as NOATTACHMENT.

The OBJECT keyword is typically used in combination with the FILE and/or DIR keywords.

- FILE** The FILE keyword specifies the local file name where the information is to be saved.
- Any path details included in the FILE value must already exist. The file itself will be created or replaced as required.
- A file name must be provided when using the keyword value OBJECT(*TEXT).
- When used with the keyword value OBJECT(*ATTACHMENT) the current attachment file will be renamed to the file name indicated.
- The FILE keyword is ignored if the keyword value OBJECT(*ATTACHMENTS) is used.
- DIR** This keyword specifies the local directory where the file is to be saved. By default the JSM instance directory is used.
- If the DIR path does not already exist it will be created.
- ENTRY** Specify the file number of the attachment to be saved. The default value is 1.
- The number assigned to each attachment is determined by the order the attachments were added to the email by the sender.
- SELECT** Select an attachment file to be saved based on the attached file name.

Examples

RDML

To save the body text of the current email to a unique file in the current directory

```
USE BUILTIN(TCONCAT) WITH_ARGS('SAVE OBJECT(*TEXT) FILE(m
```

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS

or

Save all the attachments on the current email to a directory /emailattach

USE BUILTIN(TCONCAT) WITH_ARGS('SAVE OBJECT(*ATTACHMENT
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS

RDMLX

Save the first file attached to the current email to /emailattach/abc.xml

use builtin(jsmx_command) with_args(#jsmhandle 'save object(*attachment) fi

READ

The READ command places the specified message content into a working list. The working list definition must be appropriate for the expected content. For example, if you nominate to read text content a single field is required in your working list. If you are reading anything other than text (i.e. content which includes a separator value) the working list must be defined so it can handle the maximum number of columns in the content.

The ENCODING keyword and CONTENT keyword control the content handling requirements.

Required

```
READ ----- OBJECT ----- none ----->
      *TEXT
      *ATTACHMENT
```

Optional

```
>-- CONTENT ----- *TEXT ----->
      *CSV
      *TSV
      *SV
```

```
>-- SEPARATOR ----- value ----->
      *COMMA
      *SEMICOLON
      *TAB
      *TILDE
```

```
>-- ENTRY ----- 1 ----->
      value
```

```
>-- SELECT ----- value ----->
```

```
>-- ENCODING ----- *DEFAULT ----->
      value
```

```

>-- SVROW ----- value ----->

>-- SVROWLIMIT ---- value ----->
    *NONE
    *LIST
    *AVAILABLE

>-- SVMODE ----- *NONE ----->
    *IGNORE
    *USE

>-- SVHEAD ----- value ----->

>-- NUMBERFORMAT --- *NONE -----
-|
    *DEFAULT
    *CLIENT
    *USERAGENT
    value

```

Keywords

OBJECT

This keyword specifies what part of the current message is to be read.

A value of **TEXT* indicates the body text of the message is to be read into a working list. If the message content-type does not start with *text/* or *multipart/* the command will return *NOTEXT* in the status field.

Use a value of **ATTACHMENT* to nominated a specific attachment to be read into a working list. Use this value in combination with the *ENTRY* or *SELECT* keywords to identify the attachment to be saved. If neither *ENTRY* nor *SELECT* is specified the first attachment is saved. If there are no attachments associated with the current message the command status is returned as *NOATTACHMENT*.

Refer to [RDML / RDMLX Reserved Keywords](#) for details on how to define the working list on the

command.

CONTENT

By default the email content is processed as text.

If the attachment is plain text, use `CONTENT(*TEXT)` to read the text into the working list.

Alternately the content can be processed as a Comma separated variables (`*CSV`), Tab separated variables (`*TSV`) or separated by a variable (`*SV`) as specified in the separator keyword. If processing content that includes a separator variable refer to the [SEPARATOR](#), [SVHEAD](#), [SVMODE](#), [NUMBERFORMAT](#), [TRIM](#) and [TRUNCATE](#) keywords for additional processing options.

The working list used to store the body text or attachment content must be defined with an appropriate number of columns to store the data. Refer to [RDML / RDMLX Reserved Keywords](#) for details on how to define the working list on the command.

SEPARATOR

This keyword is used by separated value services to identify the content separator.

If the `SEPARATOR` keyword is present and no `CONTENT` keyword is provided then `CONTENT(*SV)` is assumed.

Refer to [SEPARATOR](#) for more information.

ENTRY

Specify the file number of the attachment to be saved. The default value is 1. The number assigned to each attachment is determined by the order the attachments were added to the email by the sender.

SELECT

Select an attachment file to be saved based on the attached file name.

ENCODING

Specify what encoding must be applied to a bytes content to convert it to a Unicode string. The default value for the `ENCODING` keyword is `*DEFAULT`.

Refer to [ENCODING](#) for more information.

SVROW	The optional keyword SVROW is used to specify the starting row when reading records. Refer to SVROW for more information.
SVROWLIMIT	The optional keyword SVROWLIMIT is used to specify the number of rows to be read. Refer to SVROWLIMIT for more information.
SVMODE	This keyword is used by content handlers and services that process separated value data to define how to handle the inbound separated value data. Refer to SVMODE for more information.
SVHEAD	The optional keyword is used to describe the field layout of the separated value data. Refer to SVHEAD for more information.
NUMBERFORMAT	This optional keyword handles numeric strings where the decimal separator is not the decimal point character ".". Refer to NUMBERFORMAT for more information.

Examples

RDML

Using the TRUNCATE keyword ensure body content is added to the working list even if the line length exceeds the length of the field in the working list.

```
DEF_LIST NAME(#BODYLST) FIELDS(#STD_TEXT) TYPE(*WORKING
CHANGE FIELD(#JSMCMD) TO('READ OBJECT(*TEXT) SERVICE_LIST
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

Attachment files can be processed according to their content and the users knowledge of the content. For example when an email has an attachment file message01.txt with content like:

first, second, third, fourth
1, 2, 3, 4
uno, dos, tres , quatro
un , deux , trios ,

Read the content of this attachment as text (with each line of content place into a single field in a working list)

```
def_list name(#attachlst) fields(#fielda) type(*working)
```

```
use builtin(jsmx_command) with_args(#jsmhandle 'read object(*attachment) c
```

or

Read the attachment as a comma separated variables file (where the working list would be required to have four columns).

```
def_list name(#attachlst) fields(#fielda #fieldb #fieldc #fieldd) type(*working)
```

```
use builtin(jsmx_command) with_args(#jsmhandle 'read object(*attachment) c
```

DELETE

The DELETE flags the current message to be deleted from the mail server when the folder is closed. Most email clients retrieve their messages from the mail server, store the messages locally then delete them from the server.

The DELETE command works in combination with the CLOSE command.

```
DELETE ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(DELETE) TO_GET(#JSM
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 delete) to_get(#jsmxsts #jsi
```

CLOSE

The CLOSE command closes the current folder and deletes flagged messages from the POP3 mail server.

```
CLOSE ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(CLOSE) TO_GET(#JSMS'
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 close) to_get(#jsmxsts #jsn
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSA Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
        *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE *SERVICE - read service trace file.
*TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.message type (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html")

will be returned to the application in this case):

```
com.acme.property.messageType=html
```

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service and removes the temporary directory.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 service_unload) to_get(#jsi
```

5.14.4 POP3MailService Examples

Examples are supplied for the most common client programming languages:

[RDML](#)

[RDMLX](#)

[RPG](#)

5.14.5 Troubleshooting

Attempting to READ an email, if you get an error message "token count does not equal list field count at record line : nn" this indicates your working list definition does not match the information being read from the email.

- If you are reading text content (i.e. READ OBJECT(*TEXT)) a single field must be provided in the working list used as the SERVICE_LIST.
- If you are reading anything other than text content (i.e. content which includes a separator) the working list must be defined to accommodate the maximum number of columns in the content.

5.15 SMSService

The SMSService allows an application to send Short Message Service text messages (more commonly known as SMS or texting) to mobile phones anywhere in the world. Over the last few years SMS has grown into an extensively used communication tool. It is normally used for mobile phone to mobile phone messaging but the requirement to send messages generated from computers to mobile phones is very established and growing.

For the SMSService to work, the details of the message needs to be sent to an SMS gateway. This is typically a company that provides the services of sending SMS messages to telecommunication providers. The SMSService provides two approaches for sending the message to the SMS gateway.

- SMTP Protocol - in this approach the SMSService prepares and sends an email with the details of the message to an SMS gateway.
- HTTP Protocol - in this approach the SMSService forwards the message details to the SMS gateway as an HTTP POST. The HTTP transport provides you with greater security including an HTTPS option, so you might use this approach should this be required by your application.

The simpler of the two transports is the SMTP approach. The SMTP transport is fairly standard across the industry. For example, most SMS gateway service providers will require an email message to be sent to them in the following format:

Email Address: <phone-number-to-be-messaged>@<mail-domain-of-the SMS-gateway-provider>.

For example nnnnnnnnnn@streetdata.com.au where nnnnnnnnnn is the mobile phone number that you want to send the message to).

Email Subject: Containing your account ID and password (for your subscription to this SMS provider).

Email Message: Containing the message to be sent to the mobile phone.

Some SMS gateway providers may require the information to be sent to them in other formats, but with the SMTP transport approach it is relatively simple to construct your email to meet the stipulated requirements.

The HTTP transport is less flexible, and there is no industry standard amongst the providers. Some may require you to send them the HTTP POST as named-value-pairs, while other may require a SOAP request, and others some other form of XML request.

The shipped SMSService comes with "ready-to-run" HTTP solutions that work with the service provided by StreetData, Kapow, Bulker, IntelliSoftware, TextMarker and ViaNett.

Should you wish to use a HTTP transport provided by another SMS gateway service provider, please contact your LANSA vendor for advice.

The SMSService is not currently set up to handle Multimedia Messaging Server (MMS) messages, though this restriction may potentially be overcome by using the SMTPMailService.

Related Services

The SMSService is not dependant on other services.

Some service providers may provide a delivery notification response (success or failure for example) or audit on the messages that you have sent. Such messages would normally be sent back to your 'From' address. You may wish to monitor these using the POP3MailService.

In many cases the SMTPMailService will enable you to achieve the same results. The SMSService does provide you with a more flexible solution though, including the ability to send the information via HTTP.

Technical Specifications

Most of the information required for the SMSService is defined in the SMSService property file. This ensures a consistent use of the properties and avoids the need to declare this information each time the service is used.

5.15.1 What can I use the SMSService for?

SMS messaging is now a very widely used tool for communicating short pieces of text information via mobile phones.

The following provides just a few ideas on how and where the SMSService may be applied.

Send customers updates on the progress of their orders through the supply chain

Customers like to be kept up to date on how their orders are processing. An ERP system for a logistics company could send SMS messages to its customers to keep them up to date on the whereabouts of their shipment. As a shipment passes through its various ports a trigger could be set to send an SMS message to the customer.

Alert messages to engineers

Engineers in the field could be notified via an SMS message of customers who's equipment has failed and needs attending to. In addition to providing a cheaper alternative to getting the information to the engineer (as opposed to a telephone call to a mobile phone), an SMS approach also does not need to be attended to immediately. Hence, if the engineer is already focusing on another problem they may wait for a more suitable time to respond to the support request.

5.15.2 Using the SMSService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application to send an SMS to a mobile phone would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

SET

SEND

SERVICE_UNLOAD

JSM(X)_CLOSE

5.15.3 SMSService Properties

The following lists the contents of the SMSService.properties as they are shipped in a standard LANSA Integrator installation. These values are loaded as the defaults when the SERVICE_LOAD command is run.

The SMSService.properties file consists of 2 sets of key words - one set relevant to the HTTP transport, and another set relevant to the SMTP transport.

By default the HTTP transport will be used by the SMSService. To use the SMTP transport, comment out the transport=*http property to allow the transport=*smtp property to be active or add a transport=*smtp property after the transport=*http property. Alternatively, use the TRANSPORT command keyword to override the transport property value.

If you are using the HTTP or SMTP transport, then you should set up your default values in the SMSService.properties file according to the directives from your SMS gateway service provider. Alternatively, use the SET command in your application to set these values at runtime.

```
#!<studio-project id="20000000-000000" name="lansa">
#
# SMSService resources ( Default )
#
transport=*smtp
port=25
server=nnn.nnn.nnn.nnn
subject=user+password
from.address=person@mycompany.com
mail.domain=mycompany.com
mobile.domain=smscompany.com
# charset=iso-8859-1
# encoding=ISO8859_1
#
transport=*http
provider=*streetdata
uri=/admin/msg.php
host=www.streetdata.com.au
secure=*yes
account.user=12345
```

```
account.password=abcde
# sender=identifier
#
#!</studio-project>
```

The following explains the keywords defined in the SMSServices.properties file.

Keywords

transport	<p>This is used to nominate the transport mechanism you will use to send the SMS details to the SMS gateway service provider.</p> <p>There are two options:</p> <p>A value of *SMTP indicates that the SMS details is to be sent by email to the SMS gateway service provider.</p> <p>A value of *HTTP indicates that the SMS details are to be sent using HTTP.</p> <p>The transport you select will most likely be based on the service you subscribe to with the SMS gateway service provider.</p> <p>This value can be overridden in the application using the TRANSPORT keyword on the SET command.</p>
provider	<p>This keyword is only used with the *HTTP transport. This value specifies which of the SMS providers to use.</p> <p>This value can be overridden in the application using the PROVIDER keyword on the SET command.</p>
uri	<p>This keyword is only used with the *HTTP transport.</p> <p>This is the HTTP resource supplied to you by the SMS gateway service provider.</p> <p>This value can be overridden in the application using the URI keyword on the SET command.</p>
host	<p>This keyword is only used with the *HTTP transport.</p> <p>This is the host value supplied to you by the SMS gateway service provider.</p> <p>This value can be overridden in the application using this</p>

	HOST keyword on the SET command.
sender	<p>This keyword is only used with the *HTTP transport.</p> <p>This identifies the sender of the SMS message. This feature is optional as some providers do not support it.</p> <p>This value can be overridden in the application using this SENDER keyword on the SET command.</p>
secure	<p>This keyword is only used with the *HTTP transport.</p> <p>You will need to set this value to *YES to indicate that you would like to send the details using SSL (encrypted HTTPS).</p> <p>The default value at installation is *NO.</p> <p>This value can be overridden in the application using the SECURE keyword on the SET command.</p>
account.user	<p>This keyword is only used with the *HTTP transport.</p> <p>This is your account name supplied to you by your SMS gateway service provider.</p> <p>This value can be overridden in the application using the ACCOUNT_USER keyword on the SET command.</p>
account.password	<p>This keyword is only used with the *HTTP transport.</p> <p>This is the password for your account.</p> <p>This value can be overridden in the application using the ACCOUNT_PASSWORD keyword on the SET command.</p>
port	<p>This keyword is only used with the *SMTP transport</p> <p>This is the TCP/IP port that the SMTP mail server is running on.</p> <p>The default value is 25.</p> <p>This keyword is optional.</p> <p>This value can be overridden in the application using the PORT keyword on the SET command.</p>
server	<p>This keyword is only used with the *SMTP transport</p> <p>This is the address of the SMTP server you are using to send the email that contains the SMS details.</p>

This value can be overridden in the application using the SERVER keyword on the SET command.

subject

This keyword is only used with the *SMTP transport.

It will contain the subject of the email.

Typically this would contain your user account and account password (in the format 'user+password') for the SMS gateway that you are using. Confirm the format with your provider as it may have its own format.

This value can be overridden in the application using the SUBJECT keyword on the SET command.

from.address

This keyword is only used with the *SMTP transport.

This is the address to which you want email responses from the SMS gateway service provider routed back to. Such responses might include SMS status information (such as a failed SMS).

This would normally be one of your company email addresses - perhaps one that is dedicated to receiving such responses. It should be a valid email address. Many providers will only accept addresses that are already registered with them.

This value can be overridden in the application using the FROM keyword on the SET command.

mail.domain

This keyword is only used with the *SMTP transport.

This is the mail domain as issued by the SMTP HELO command. This value tells the SMTP server that this is your mail domain.

This keyword is optional.

This value can be overridden in the application using the MAILDOMAIN keyword on the SET command.

mobile.domain

This keyword is only used with the *SMTP transport.

This is the mail domain to which you will send the email with the SMS details.

Normally the mobile phone number you are sending the

SMS to will prefix this value. So for example, if you are sending the SMS to a number 12345678 and the mobile domain is streetdata.com.au, then the full email address that the message will be sent to is 12345678@streetdata.com.au.

This value can be overridden in the application using the MOBILEDOMAIN keyword on the SET command.

charset

This keyword is only used with the *SMTP transport.

This is the character set encoding of the body text and subject.

This keyword is optional.

This value can be overridden in the application using the CHARSET keyword on the SET command.

encoding

This keyword is only used with the *SMTP transport.

This is the body text encoding

This keyword is optional.

This value can be overridden in the application using the ENCODING keyword on the SET command.

5.15.4 SMSService Commands

Your application issues commands to the SMSService by passing the command strings through the Java Services Manager using the JSM_COMMAND or JSMX_COMMAND Built-In Function or an API for your chosen development language.

The commands that the SMSService processes are:

SERVICE_LOAD

SET

SEND

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
---->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

SERVICE The name of the service to be loaded - in this case SMSService.

TRACE To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.

The possible values for the TRACE keyword are:

- *NO
- *YES
- *ERROR

The trace option *ERROR will turn on tracing and if the

service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.

TRACE_NAME This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(SMSSERVICE)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jsmcommand := 'service_load service(SMSService)'
use builtin(jsmx_command) with_args(#jsmxhdl1 #jsmcommand) to_get(#jsn
```

SET

The SET command is an optional command and is used primarily to override the values defined in the SMSService.properties file.

To set multiple keywords inside your application you may either use the SET command multiple times, or use one SET command with multiple keywords defined in it.

Optional

```
SET ----- TRANSPORT ----- *HTTP -----
>
    *SMTP

>-- PROVIDER ----- *KAPOW ----->
    *VIANETT
    *BULKER
    *STREETDATA
    *TEXTMARKETER
    *INTELLISOFTWARE

>-- HOST ----- value ----->

>-- SENDER ----- value ----->

>-- SECURE ----- *NO ----->
    *YES

>-- URI ----- value ----->

>-- ACCOUNT_USER ---- value ----->

>-- ACCOUNT_PASSWORD- value -----
>

>-- SERVER ----- value ----->

>-- PORT ----- value ----->
```

```
>-- MAILDOMAIN ----- value ----->
>-- MOBILEDOMAIN ---- value ----->
>-- FROM ----- value ----->
>-- CHARSET ----- value ----->
>-- ENCODING ----- value ----->
>-- SUBJECT ----- value -----|
```

Keywords

TRANSPORT

This is used to nominate the transport mechanism you will use to send the SMS details to the SMS gateway service provider.

There are two options:

*SMTP indicates that the SMS details are to be sent by email to the SMS gateway service provider.

*HTTP indicates that the SMS details are to be sent by HTTP to the SMS gateway service provider. The transport you select will likely be based on the service you subscribe to with the SMS gateway service provider.

This keyword can be used to override the **transport** value in the SMSService.properties file.

PROVIDER

This keyword is only used with the *HTTP transport.

You will only use it if you have selected *HTTP as your transport.

This keyword can be used to override the **provider** value in the SMSService.properties file.

HOST	<p>This keyword is only used with the *HTTP transport.</p> <p>This is the host value supplied to you by the SMS gateway service provider.</p> <p>This keyword can be used to override the host value in the SMSService.properties file.</p>
SENDER	<p>This keyword is only used with the *HTTP transport.</p> <p>This identifies the sender of the SMS message.</p> <p>This feature is optional as some providers do not support it.</p> <p>This keyword can be used to override the sender value in the SMSService.properties file.</p>
SECURE	<p>This keyword is only used with the *HTTP transport.</p> <p>You will set this value to *YES to indicate that you would like to send the details using SSL (encrypted HTTPS).</p> <p>The default value at installation is *NO.</p> <p>This keyword can be used to override the secure value in the SMSService.properties file.</p>
URI	<p>This keyword is only used with the *HTTP transport.</p> <p>This is the HTTP resource supplied to you by the SMS gateway service provider.</p> <p>This keyword can be used to override the uri value in the SMSService.properties file.</p>
ACCOUNT_USER	<p>This keyword is only used with the *HTTP transport.</p> <p>This is your account name supplied to you by your SMS gateway service provider.</p> <p>This keyword can be used to override the account.user value in the SMSService.properties file.</p>

ACCOUNT_PASSWORD	<p>This keyword is only used with the *HTTP transport.</p> <p>This is the password for your account.</p> <p>This keyword can be used to override the account.password value in the SMSService.properties file.</p>
SERVER	<p>This keyword is only used with the *SMTP transport</p> <p>This is the address of the SMTP server you are using to send the email that contains the SMS details.</p> <p>This keyword can be used to override the server value in the SMSService.properties file.</p>
PORT	<p>This keyword is only used with the *SMTP transport.</p> <p>This is the TCP/IP port that the SMTP mail server is running on.</p> <p>The default value is 25.</p> <p>This keyword is optional.</p> <p>This keyword can be used to override the port value in the SMSService.properties file.</p>
MAILDOMAIN	<p>This keyword is only used with the *SMTP transport.</p> <p>This is the mail domain as issued by the SMTP HELO command. This value tells the SMTP server that this is your mail domain.</p> <p>This keyword is optional.</p> <p>This keyword can be used to override the mail.domain value in the SMSService.properties file.</p>
MOBILEDOMAIN	<p>This keyword is only used with the *SMTP transport.</p>

This is the mail domain to which you will send the email with the SMS details.

Normally the mobile phone number you are sending the SMS to will prefix this value. So for example, if you are sending the SMS to a number 12345678 and the mobile domain is streetdata.com.au, then the full email address that the message will be sent to is 12345678@streetdata.com.au.

This keyword can be used to override the **mobile.domain** value in the SMSService.properties file.

FROM

This keyword is only used with the *SMTP transport.

This is the FROM address of the email that you want emails responses from the SMS gateway service provider routed back to. Such responses might include SMS status information (such a failed SMS).

This would normally be one of your company email addresses - perhaps one that is dedicated to receiving such responses. It should be a valid email address. Many providers will only accept values that are already registered with them.

The value supplied will be appended to the mail domain so you only need to enter the first part of the address. For example, if the mail domain is mycompany.com and you supply a FROM value of john.smith then full return email address will be john.smith@mycompany.com.

This keyword can be used to override the **from.address** value in the SMSService.properties file.

CHARSET

This keyword is only used with the *SMTP

transport.

This is the character set encoding of the body text and subject.

This keyword is optional.

This keyword can be used to override the **charset** value in the SMSService.properties file.

ENCODING

This keyword is only used with the *SMTP transport.

This is the body text encoding

This keyword is optional.

This keyword can be used to override the **encoding** value in the SMSService.properties file.

SUBJECT

This keyword is only used with the *SMTP transport.

This will contain the subject of the email.

Typically this would contain your user account and account password (in the format user+password) for the SMS gateway that you are using. Do confirm with your provider as they may have their own format.

This keyword can be used to override the **subject** value in the SMSService.properties file.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET ACCOUNT_NAME(
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'set account_name(userid) :
```

SEND

```
SEND ----- TO ----- value ----->
      >-- MSG ----- value -----|
```

Keywords

TO This should contain only the telephone number of the mobile phone you wish to send the SMS message to.

For example, +6091234567

MSG This should contain the message that you wish to send.

Comments / Warnings

The SMS gateway service provider that you are working with may have restrictions on the length of the message that can be sent to a mobile phone. The SMSService will pass whatever information you place in the MSG keyword to the provider, but it is your responsibility to ensure that the amount of data can be handled by the provider. Your provider will be able to advise you on any restrictions that they might have.

For sending messages to international mobile phones, the format of the number in the TO keyword will generally be a plus sign (+), followed by the country code, followed by the phone number itself. For messages to local numbers, generally the country code is not required although it will still work if you include it. You should confirm with your provider exactly what the required format is.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SEND TO(+6012345678)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'send to(+6012345678) msg;
```


SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service and removes the temporary directory.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 service_unload) to_get(#jsi
```

5.15.5 SMSService Examples

Tip: To test the *SMTP transport of this service, use a local email address for your mobile domain keyword. This way you can test the general flow of your application by sending emails to yourself prior to testing against the SMS gateway provider's system. Testing against the SMS gateway provider's system will cost you money – unless they offer you some free testing SMSs.

[RDML](#)

[RDMLX](#)

[RPG](#)

5.16 XMLFileService

Service Name: **XMLFileService**

The XMLFileService allows XML files to be read and created.

The XMLFileService supports the following commands:

5.16.1 SERVICE_LOAD

5.16.2 SERVICE_GET

5.16.3 READ

5.16.4 WRITE

5.16.5 SEND

5.16.6 RECEIVE

5.16.7 DELETE

5.16.8 RENAME

5.16.9 SET

5.16.10 LIST

5.16.11 SERVICE_UNLOAD

5.16.1 SERVICE_LOAD

On the SERVICE_LOAD event the service determines if schema validation is enabled.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	SCHEMA		Optional. See SCHEMA .
	VALIDATING		Optional. See VALIDATING .
	DOMSET		Optional. See DOMSET .
	DOMSETMODEL		Optional. See DOMSETMODEL .
	DOMSETRESULT		Optional. See DOMSETRESULT .
	DOMGET		Optional. See DOMGET .
	DOMGETRESULT		Optional. See DOMGETRESULT .

5.16.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.16.3 READ

When the READ command of this service is executed the following steps occur:

If the HOST keyword is not present then the contents of the local file is read, else the FTP get protocol is used to read the remote file.

The service determines the encoding to apply to the received byte content to convert it to Unicode content. The default action is to auto-detect the encoding of the XML source.

The service converts the Unicode content to a UTF-8 byte content.

The service determines if the Unicode data needs to be archived.

To read a file from the local file system, only requires the FILE keyword.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
READ	FILE	value	Mandatory. File Path.
	ENCODING		Optional. See ENCODING . Default is to auto-detect encoding.
	ARCHIVE		Optional. See ARCHIVE .

To read a file from a remote file system using the FTP protocol requires the HOST keyword.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
READ	HOST	value	Conditional. FTP server.
	FILE	value	Mandatory. File Path.
	ENCODING		Optional. See ENCODING . Default is to auto-detect encoding.

ARCHIVE		Optional. See ARCHIVE .
USER	value	Mandatory. User.
PASSWORD	value	Mandatory. Password.
NAMEFMT	0	Optional. Path name format. library/file.member.
	1	Path name format. /directory/directory/file.
DATALINK	*PASV	Optional. Default. Data channel connection mode.
	*PORT	Data channel connection mode.
MODE	*BINARY	Optional. Default. Data transfer mode.
	*ASCII	Data transfer mode.

5.16.4 WRITE

When the WRITE command of this service is executed the following steps occur:

The service determines if the output Unicode content needs to be archived.

The service determines the encoding to apply to the output Unicode content to convert it to byte content.

If the HOST keyword is not present the byte content is written to a local file, else the FTP put protocol is used to write the remote file.

To write a file to the local file system, only requires the FILE keyword.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
WRITE	FILE	value	Mandatory. File Path.
	ENCODING		Optional. See ENCODING . Default encoding is UTF-8.
	ARCHIVE		Optional. See ARCHIVE .

To write a file to a remote file system using the FTP protocol requires the HOST keyword.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
WRITE	HOST	value	Conditional. FTP server.
	FILE	value	Mandatory. File Path.
	ENCODING		Optional. See ENCODING . Default encoding is UTF-8.
	ARCHIVE		Optional. See ARCHIVE .
	USER	value	Mandatory. User.

PASSWORD	value	Mandatory. Password.
NAMEFMT	0	Optional. Path name format. library/file.member.
	1	Path name format. /directory/directory/file.
DATALINK	*PASV	Optional. Default. Data channel connection mode.
	*PORT	Data channel connection mode.
MODE	*BINARY	Optional. Default. Data transfer mode.
	*ASCII	Data transfer mode.

5.16.5 SEND

When the SEND command of this service is executed the following steps occur:

The service using the XSL keyword transforms the program fields and list into the resultant XML. If a FRAGMENT keyword is present this transformation is stored, as this is only a fragment of the final XML to be sent. When no FRAGMENT keyword is present this transformation is treated as the last and the final output XML is constructed and ready to be written to a file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	XSL		Mandatory. See XSL .
	FRAGMENT		Optional. See FRAGMENT .
	OUTPUT		Optional. See OUTPUT .

5.16.6 RECEIVE

When the RECEIVE command of this service is executed the following steps occur:

If there is no content, then an OK status is returned.

The service then transforms the XML and binds the data into the program fields and list.

The receive command can be called more than once to receive data from a complex XML document.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

RECEIVE	XSL		Mandatory. See XSL .
---------	-----	--	--------------------------------------

5.16.7 DELETE

The DELETE command deletes the specified file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
DELETE	FILE	value	Mandatory. File Path.

5.16.8 RENAME

The RENAME command renames the specified local file to a new name. The TO file must be in the same directory as the original, FROM file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RENAME	FROM	value	Mandatory. Old file name.
	TO	value	Mandatory. New file name.
	REPLACE	*YES	Optional. Replace file.
		*NO	Do not replace file. The default value of the REPLACE keyword is *NO. If the TO filename exists, then the rename process is stopped, unless the REPLACE keyword is *YES.

5.16.9 SET

The SET command sets the current working directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	DIR	value	Mandatory. Directory.

5.16.10 LIST

The LIST command will fill the command's list object with absolute file names. A single field working list is required to receive the canonical file paths.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value	
LIST	DIR	value	Optional. Fill working list with directory listing. All files in that directory will be selected.	
		EXT	value	Optional. Filtering on file extension. The filtering match is case insensitive.
		SORT	*NONE	Optional. Allows sorting of file names. Default. No sorting.
		*NAME	Sort on file name.	
		*MODIFIED	Sort on modified date.	
	REVERSE	*YES	Optional. Reverse the order of the sort.	
	*NO	Default.		

If a directory path specified by the DIR keyword is used then filenames in that directory will be selected.

Example

```
LIST DIR(/xmldata) EXT(XML) SERVICE_LIST(PATH)
```

This will return a list of filenames in the /xmldata directory that have an XML extension.

5.16.11 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_UNLOAD			

5.17 XMLBindFileService

The XMLBindFileService is designed to make the job of reading data from XML documents into applications and the creation of XML documents by applications simple. This is one of several services that are designed to consume the classes generated from the XML Binding Wizard (a feature of the LANSA Integrator Studio), so the use of LANSA Integrator Studio is a key part of the XMLBindFileService.

It is important to note that this service uses these bindings created with LANSA Integrator Studio and not XSL stylesheets. Because of this, this service is far simpler to use than using one of the services that requires an XSL stylesheet, so in general you would use this over the other options.

So, for example, if you receive an order in XML format into a directory on your server, the XMLBindFileService could be used to read the order data in the XML document, convert the data to fields inside your program, from which the program can then process the data as required. Alternatively, your application might be the one that needs to create an order to be sent to a supplier. In which case your application would prepare the data required for the order then use the XMLBindFileService to create the XML document.

It is important to be aware that the XMLBindFileService is not responsible for the transportation of the XML file between the client and the server. In the event of using it to READ an XML file into a program, it assumes that the file is already there. If you are receiving an XML document then it is generally the sender's responsibility to send the file to you, though you will of course need to tell them how and where to do that. Having said this, there may be times when you may need to 'pull' the XML document from another location to your server for a READ into your application. When this service is used to WRITE, or create, an XML document it is not concerned with how, when or where it is delivered to the recipient. If you are the one creating the XML document, then more often than not it would be your responsibility to send the document.

If you are responsible for the transportation of the XML document then you might want to consider using one of a range of transport services that are also available with LANSA Integrator. Refer to the Related Services for further information on these transport based services.

Related Services

The XMLBindFileService is one of several services that can consume the classes generated from the XML Binding Wizard (a feature of the LANSA

Integrator Studio). Other services in this family combine the XML binding with specific transport support. These include:

- XMLBindQueueService
- HTTPInboundXMLBindService
- HTTPOutboundXMLBindService

The XMLBindFileService does not provide transport of the XML document. If you need to transport the document, you may need to use one of the services above, or combine the XMLBindFileService with one of the transport focused services such as:

- FTPService - this service allows an application to send and receive files to and from a remote FTP server.
- HTTPService - this service provides a number of content handlers to send and receive content using the HTTP protocol.
- SMTPMailService - this service could be used to attach an XML document to an email and send it to a recipient.
- SMTPAttachmentSignatureService - this service could be used to send the XML document by email attachment with a digital signature.
- POP3MailService - this service could be used to receive an email that contains an XML document as an attachment.
- JMSFileService - this service could be used to send and receive the XML documents using enterprise messaging systems such as ActiveMQ, SonicMQ, TibcoMQ and WebSphereMQ.

The XMLParserService and XMLFileService provide alternate means of reading and writing XML files, but for most new applications, using one of the services associated with the XML Binding Wizard (such as the XMLBindFileService) is the recommended approach. The XMLParserService and XMLFileService require the use of XSL stylesheets, which is a far more complex approach than using the XML Binding Wizard approach.

Technical Specifications

The XMLBindFileService needs to be used in conjunction with the bindings created using the XML Binding Wizard of the LANSAs Integrator Studio. You must start with a sample XML document to use to describe the mappings between the XML elements and the fields in your program. The XML document may be created by yourself or supplied by a third party. Once created you will

need to move the relevant .jar and property files to the server before you can run the application.

Please refer to the XML Binding Wizard if you are not familiar with this.

5.17.1 What can I use the XMLBindFileService for?

To receive an order in XML format and send a reply

You may be receiving orders from a distributor in XML document format, so you need a way to transform the data held within the document to information that is useful to your application - specifically, fields and lists (in the case of LANSAs applications), and subfiles (in the case of RPG, for example). As a first step in the exercise you would use the [XML Binding Wizard](#), in LANSAs Integrator Studio, to bind the XML elements to field names that are used in your application. Once you have done this you would use the XMLBindFileService in your application to read the data into application fields and lists (using the READ and GET commands). After that has been completed, your application can then do what it needs to in order to process the order into your system.

In many cases, you will need to send some form of acknowledgement back to the sender. This time, the XMLBindFileService can be used to create an XML document as a response, specifically using the WRITE and SET commands that come with this service.

How would you handle a situation where you have received many XML documents into a directory? You could use the LIST command that comes with the XMLBindFileService. This will supply you with a list of all the files that are in the directory, and you can then use that list to process the files that you want to process.

Passing Data Between Internal Systems

Many companies face the problem of how to integrate data from disparate systems. They might be running Oracle financials on a Windows server, and have a home grown RPG or LANSAs system running on an IBM i server, and need to move data between the two systems. XML is becoming the preferred approach for the movement of such data. Since this is within your organisation, the job is easier for you, as you control both sides of the equation. Oracle might have its own way of generating the XML data that you are after, and if it does not, you could use the XMLBindFileService to transform the data you need moved into an XML document using the SET and WRITE commands and place it on the network drive. Once created, an application on the IBM i could be kicked into action to READ the generated XML documents so that the data can be made available to the IBM i.

5.17.2 Using the XMLBindFileService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you must complete the same basic steps.

For example, an application that needs to READ an XML document would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

READ

BIND

GET

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Whereas, an application that needs to WRITE to an XML document would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

BIND

SET

WRITE

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Refer to the [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.17.3 XMLBindFileService Properties

Properties that affect the operation of the XMLBindFileService are specified in the XMLBindFileService.properties file. The properties of interest fall into two categories:

- Properties in the form `service.xxxxx=` that associate the service name that you specify in the BIND service command with the classes generated from the XML Binding Wizard. These properties are generated by the XML Binding Wizard when you build your project, so you should not normally have to manually enter them. You may have to deploy them to your production environment however – refer to [XML Binding Wizard](#) for more information.
- General properties that provide default values affecting the operation of the service. These are described below:

The default properties file is relatively uninteresting and the bulk of it is taken up with messages. The following explains the keywords that do come with the default XMLBindFileService.properties file.

Keywords

- | | |
|--------------------------------|---|
| <code>validation.schema</code> | This keyword is used to turn on or off XML schema validation |
| <code>validation.parser</code> | This keyword is used to turn on or off XML validation. |
| <code>systemid.file</code> | This keyword is used to specify any DTD Entity Resolves you may wish to include. You may specify multiple DTD Entity Resolves here. |

5.17.4 XMLBindFileService Commands

Your application issues commands to the XMLBindFileService by passing the command strings through the Java Services Manager using the JSM_COMMAND or JSMX_COMMAND built-in function or an API for your chosen development language.

The commands that the XMLBindFileService processes are:

SERVICE_LOAD
READ
WRITE
BIND
GET
SET
DELETE
RENAME
LIST
CLOSE
SERVICE_GET
SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
---->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name ----->  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

```
>-- SCHEMA ----- *NO ----->  
      *YES
```

```
>-- VALIDATING ----- *NO ----->  
      *YES
```

```
>-- DOMSET ----- *READER -----|  
      *STREAM
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

SERVICE	The name of the service to be loaded - in this case XMLBindFileService.
---------	---

TRACE	<p>To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.</p> <p>The possible values for the TRACE keyword are:</p> <ul style="list-style-type: none"> • *NO • *YES • *ERROR <p>The trace option *ERROR will turn on tracing and if the service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.</p>
TRACE_NAME	<p>This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory.</p> <p>Special keyword values are also available for the TRACE_NAME keyword.</p> <ul style="list-style-type: none"> • *SERVICE • *PROCESS • *FUNCTION • *JOBNAME • *JOBUSER • *JOBNUMBER
SCHEMA	Optional. See SCHEMA .
VALIDATING	Optional. See VALIDATING .
DOMSET	Optional. See DOMSET .

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(XMLBINDI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jsmcommand := 'service_load service(XMLBindFileService)'
use builtin(jsmx_command) with_args(#jsmxhdl1 #jsmcommand) to_get(#jsn
```


READ

The READ command is used to parse and load XML document.

When the READ command is executed, the following steps occur:

1. The service determines the encoding to apply to the received byte content (of the XML document) to convert it to Unicode content. The default action is to auto-detect the encoding of the XML source. Autodetect will determine the encoding from the XML declaration encoding, for example `<?xml version="1.0" encoding="UTF-8"?>`.
2. The service converts the Unicode content to a UTF-8 byte content and parses the content into a document object.
3. The service determines if the Unicode data needs to be archived.

```
                                Required
READ ----- FILE ----- file path ----->
                                Optional
>-- ENCODING ----- value ----->
>-- ARCHIVE ----- value -----|
```

Keywords

FILE This keyword is used to specify the file name and path of the XML document.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

- Absolute path
For example:
/orders/order.xml
C:/orders/order.xml
C:\orders\order.xml

or

- Relative path.
For example, `orders/order.xml` (note, no '/' at the start), in which case the `order.xml` document must reside in the `orders` directory under the JSM Instance directory on your server.

For the IBM i you can specify:

- Absolute path.
For example:
`/orders/order.xml`

or

- Relative path.
For example, `orders/order.xml` (note, no '/' at the start), in which case the `order.xml` document must reside in the `orders` directory under the JSM Instance directory on your server.

Note: Whatever directory structure you specify must already exist.

This keyword is mandatory.

ENCODING Refer to **ENCODING** for more complete information on this keyword.

The default value for the `XMLBindFileService` is `*AUTODETECT`.

This keyword is optional.

ARCHIVE Refer to **ARCHIVE** for more complete information on this keyword.

This keyword is optional.

Comments / Warnings

The **ARCHIVE** keyword is a very useful way to store away XML documents after they have been processed.

Examples

RDML

* Define the fields used by the JSM Commands

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

```
CHANGE FIELD(#JSMCMD) TO('READ FILE(orders/order.xml) ARCHIVE
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

* Define the fields used by the JSM Commands

```
Define FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
Define FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
Define FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
Define FIELD(#JSMHND) TYPE(*CHAR) LENGTH(4)
```

```
#JSMCMD := 'READ FILE(orders/order.xml) ARCHIVE(archive/arc_order.xi
Use BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMHND #JSMCMD) T
```

WRITE

The WRITE command serializes the document object out to a specified file. It is used as the final step in creating an XML document from fields set inside your program. You will use the SET command one or more times prior to using the WRITE command to set up the data in the document.

Required

WRITE ----- *FILE* ----- *file path* ----->

Optional

>-- *INDENT* ----- **NO* ----->
 **YES*

>-- *IDENT-AMOUNT* ----- *value* ----->

>-- *DOCTYPE* ----- *value* ----->

>-- *PUBLIC* ----- *value* ----->

>-- *OMIT-DECLARATION* --- **NO* ----->
 **YES*

>-- *ENCODING* ----- *value* ----->

>-- *BINDTRACE* ----- **NO* ----->
 **YES*

>-- *FILTER* ----- *value* -----|

Keywords

FILE

This keyword is used to specify the file name and path of the XML document to be created.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

Absolute path.

For example:

/orders/order.xml

C:/orders/order.xml

C:\orders\order.xml

or

Relative path.

For example, orders/order.xml (note, no '/' at the start), in which case the document order.xml will be placed in the orders directory under the JSM Instance directory on your server.

For the IBM i you can specify:

Absolute path.

For example:

/orders/order.xml

or

Relative path.

For example, orders/order.xml (note, no '/' at the start), in which case the document order.xml will be placed in the orders directory under the JSM Instance directory on your server.

Note: The directory structure must exist.

This keyword is mandatory.

INDENT

This keyword is used to specify whether or not you would like the XML content to be indented.

There are two options:

***YES** - to indicate that you want the XML content to be indented.

***NO** - to indicate that you do not want the XML content indented.

- The default value is *NO.

This keyword is used in conjunction with the INDENT-AMOUNT keyword.

This keyword is optional.

INDENT-AMOUNT

This keyword is used to specify the number of spaces to use for the indentation.

The default value is 0 (which is akin to having indenting turned off).

If the INDENT keyword is set to *NO then this keyword will have no effect.

This keyword is optional.

DOCTYPE

This keyword is used to specify the system component of the document type declaration.

Whatever value you specify here will be placed into the document as the document type declaration. It will be placed just after the XML declaration (if this is included).

This keyword is optional.

PUBLIC

Conditional. The optional public component of the DOCTYPE declaration.

OMIT-DECLARATION

This keyword is used to specify whether you want to include the XML declaration or not.

There are two options:

***YES** - to indicate that you want the XML declaration omitted.

***NO** - to indicate that you want the XML declaration to remain included.

The default value is *NO.

This keyword is optional.

ENCODING

Optional. See [ENCODING](#).

Default encoding is UTF-8.

BINDTRACE

This keyword is used to turn on tracing for the outbound bind result.

There are two options:

***YES** - to switch tracing on.

***NO** - to switch tracing off.

This keyword is optional.

FILTER

Refer to [FILTER](#) for more complete information on this keyword.

This keyword is optional.

Examples

RDML

* Define the fields used by the JSM Commands

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

```
CHANGE FIELD(#JSMCMD) TO('WRITE FILE(response/rsp_order.xml) IN  
AMOUNT(1)')
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

* Define the fields used by the JSM Commands

```
Define FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
Define FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
```

```
Define FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

```
Define FIELD(#JSMHND) TYPE(*CHAR) LENGTH(4)
```

```
#JSMCMD := 'WRITE FILE(response/rsp_order.xml) INDENT(*YES) INDEI  
AMOUNT(1)'
```

```
Use BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMHND #JSMCMD) T
```

BIND

The BIND command binds the specified service code to the loaded inbound document or creates a new empty outbound document.

The service specified as a part of the command will have been created using the XML Binding Wizard in LANSa Integrator Studio.

You will need to specify the type of bind as outbound or inbound - a READ is normally associated with an inbound type while a WRITE is normally associated with an outbound type.

Required

```
BIND ----- SERVICE ----- value ----->
```

```
>-- TYPE ----- *INBOUND ----->  
                  *OUTBOUND
```

Optional

```
>-- BINDTRACE ----- *NO ----->  
                  *YES
```

```
>-- FILTER ----- value -----|
```

Keywords

SERVICE This keyword is used to in an XMLBindFileService property lookup using service.'value' and service.archive.'value' to locate the binding class and binding jar file to be used. The value here will be the corresponding service class name that was defined in the XML Binding Wizard.

This keyword is mandatory.

TYPE This keyword is used to specify the type of bind to be performed, and it will depend on whether you are intending to perform a READ of an existing file or a WRITE of a new file.

There are two possible values:

***INBOUND** - using this value will bind the specified service code to the inbound document loaded as a part of the preceding READ command.

***OUTBOUND** - using this value will create a new empty outbound document.

This keyword is mandatory.

BINDTRACE This keyword is used to turn on tracing for the inbound bind result.

There are two options:

***YES** - to switch tracing on.

***NO** - to switch tracing off.

This keyword is optional.

FILTER Refer to [FILTER](#) for more complete information on this keyword.

This keyword is optional.

Comments / Warnings

The position of the BIND command in the code depends on whether you are intending to read data from an existing XML document or you are intending to create a new XML document. If you are reading data from an existing XML document, the code flow will look something like this:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

READ

BIND

GET (one or more times)

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Whereas, an application that needs to WRITE to an XML document would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

BIND
SET (one or more times)
WRITE
CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Examples

RDML

* Define JSM fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)  
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)  
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Bind

```
CHANGE FIELD(#JSMCMD) TO('BIND SERVICE(INBOUNDORDER) TY  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

* Define the fields used by the JSM Commands

```
Define FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)  
Define FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)  
Define FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)  
Define FIELD(#JSMHND) TYPE(*CHAR) LENGTH(4)
```

```
#JSMCMD := 'BIND SERVICE(INBOUNDORDER) TYPE(*INBOUND) BII  
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

GET

The GET command is used to read the data from the inbound document object. The GET command is integral part of loading data from an XML document into your program. It must be preceded by the READ and the BIND commands.

Conditional

```
GET ----- LIST ----- value ----->
```

```
>-- FRAGMENT ----- value ----->
```

```
>-- INSTRUCTION --- value -----|
```

Keywords

LIST This keyword is used to get a list from the loaded XML document.

Refer to the Lists and Variables section below for further details on how to use this keyword.

This keyword is conditional.

FRAGMENT This keyword is used to get a field fragment from the loaded XML document.

Refer to the Lists and Variables section below for further details on how to use this keyword.

This keyword is conditional.

INSTRUCTION This keyword is used to get the XML processing instructions.

This keyword is optional.

Comments / Warnings

You can only use one keyword at a time.

There are two ways in which you can read a list from an XML document.

1. You can either define the relevant section of the XML code as a list (using the XML Binding Wizard), and use the LIST keyword in your GET

command. This will retrieve the entire list in one go into a working list.

2. Alternatively, you can define the relevant section as a collection of fragments (using the XML Binding Wizard) and then use the FRAGMENT keyword in your GET command and place this in a loop.

See the Examples for details of how to do this.

Note: Fragments and Lists

The following points are important when using GET FRAGMENT and GET LIST together.

- When you issue a GET FRAGMENT, that fragment becomes the current fragment. That is, the fragment pointer moves to the current one.
- You need to read the fragments in an order that allows all fragments to be accessed. You can only access lists and child fragments once you have positioned the fragment pointer to its parent fragment. So, in a way, fragments are a bit like branches on a tree - to access lists and fragments further down the tree you will need to position the fragment pointer to the parent branch.
- For example, when you issue a GET FRAGMENT, a GET LIST can only access lists within this current fragment. Therefore, you should read all the lists (using GET LIST) in the current fragment before you move on to another. Once you move onto another fragment, then the lists in the previous fragments will not be accessible. You may access the lists within a specific fragment in any order.
- When a fragment becomes the current fragment it is marked as used, so once you leave it, it will not become the current fragment again.

Lists and Variables

If you are using the LIST keyword to return a list from the loaded document into your program, you will need to ensure the following steps are taken in your program.

1. Define a working list that contains the fields that you are expecting from the XML document.
2. Use the SERVICE_LIST keyword with the LIST keyword in the GET command. The service list value should include the names of the fields in your working list without the '#'. The order of the fields should be defined here as they appear in the working list.
3. In the TO_GET portion of the JSM_COMMAND Built-In Function, include

the name of the working list that will hold the values returned.

If you are using the FRAGMENT keyword to return a field or group of fields, then you will need to add the SERVICE_EXCHANGE keyword with a value of '*FIELD'.

See the Examples for details on how to do this.

Examples

RDML

* Define JSM fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Define a list to hold the order line details

```
DEF_LIST NAME(#WRKLNES) FIELDS(#LINNUM #PARTNUM #PARTI
```

* Get SalesOrder Details

```
CHANGE FIELD(#JSMCMD) TO('GET FRAGMENT(SALESORDER) SER
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

* Get Lines

```
CHANGE FIELD(#JSMCMD) TO('GET LIST(LINE) SERVICE_LIST(LINN
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

INSTRUCTION keyword example

If the XML processing instructions were as per the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<?Label SLBK|PROFILE|208|SUCCESS?>
<Orders xmlns="here" here="yes">
```

Then you could use the following code to retrieve the instructions.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET INSTRUCTION(La
```

In this example, the instruction would be placed into the #JSMMSG field.

RDMLX

* Define the fields used by the JSM Commands

Define FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)

Define FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)

Define FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)

Define FIELD(#JSMHND) TYPE(*CHAR) LENGTH(4)

#JSMCMD := 'GET FRAGMENT(CUSTOMER) SERVICE_EXCHANGE(*F

Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get

SET

The SET command is used for two distinct operations. Use the DIR keyword to set the current working directory or alternatively use the LIST, FRAGMENT, or INSTRUCTION keywords to populate the outbound document object.

Once you have completed your SET commands your next command would invariably be the WRITE command.

Conditional

```
SET ----- LIST ----- value ----->
```

```
>-- FRAGMENT ----- value ----->
```

```
>-- INSTRUCTION --- value ----->
```

```
>-- DATA ----- value ----->
```

```
>-- DIR ----- value -----|
```

Keywords

LIST This keyword is used to place a list of values into an XML document.
Refer to the Lists and Variables section below for further details on how to use this keyword.
The LIST keyword does not have any relationship with the DIR keyword.
This keyword is conditional.

FRAGMENT This keyword is used to place a field or group of fields into an XML document.
Refer to the Lists and Variables section below for further details on how to use this keyword.
The FRAGMENT keyword does not have any relationship with the DIR keyword.
This keyword is conditional.

INSTRUCTION	<p>This keyword is used to set the XML processing instruction. This keyword is used in conjunction with the DATA keyword.</p> <p>The INSTRUCTION keyword does not have any relationship with the DIR keyword.</p> <p>This keyword is conditional.</p>
DATA	<p>This keyword is used to set the XML processing instructions data.</p> <p>This keyword is conditional. It is used in conjunction with the INSTRUCTION keyword.</p>
DIR	<p>This keyword is used to nominate a relative or absolute path to be set as the current directory.</p> <p>The DIR keyword does not have any relationship with the other keywords for the SET command.</p> <p>This keyword is conditional.</p>

Comments / Warnings

There are two possible ways to add a list of data when creating an XML document.

1. The most obvious way is to use the SET command described here with the LIST keyword, then specifying the array in a working list. With this approach the section of XML would need to be defined as a list in the XML Binding Wizard.
2. Alternatively, the same could be achieved by using the FRAGMENT keyword inside some looping code. Each loop would add a new row to the list. XML documents and readers will automatically pick up this repeating sequence as a list. With this approach the section of XML would need to be defined as a fragment in the XML Binding Wizard.

Whichever way you choose, you will need to carefully decide whether the relevant section of XML is defined as a List or a Fragment.

See the Examples following how to do this.

Lists and Variables

If you are using the LIST keyword to create a list in an XML document, you will need to ensure the following steps are taken in your program.

1. Define a working list that contains the fields that will be passed to the XML document.
2. Use the SERVICE_LIST keyword with the LIST keyword in the SET command. The service list value should include the names of the fields in the working list without the '#'. The order of the fields should be defined here as they appear in the working list.
3. In the TO_GET portion of the JSM_COMMAND Built-In Function, include the name of the working list that will hold the values to be placed in the XML document.

If you are using the FRAGMENT keyword to add a field or group of fields, then you will need to add the SERVICE_EXCHANGE keyword with a value of '*FIELD'. How to do this is shown in the Examples following.

Examples

RDML

* Define JSM fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Define Order Line fields

```
DEFINE FIELD(#LINNUM) TYPE(*DEC) LENGTH(003) DECIMALS(0) C
DEFINE FIELD(#PARTNUM) TYPE(*DEC) LENGTH(003) DECIMALS(0)
DEFINE FIELD(#PARTDSC) TYPE(*CHAR) LENGTH(030) COLHDG('Des
DEFINE FIELD(#PARTAMT) TYPE(*DEC) LENGTH(010) DECIMALS(2)
DEFINE FIELD(#PARTQTY) TYPE(*DEC) LENGTH(003) DECIMALS(0) C
```

* Define the list to hold the order lines

```
DEF_LIST NAME(#WRKLINE) FIELDS(#LINNUM #PARTNUM #PARTI
```

* Set customer details

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET FRAGMENT(CUSTO
```

* Set SalesOrder header details

```
CHANGE FIELD(#JSMCMD) TO('SET FRAGMENT(SALESORDER) SERV
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
CHANGE FIELD(#JSMCMD) TO('SET LIST(LINE) SERVICE_LIST(LINNUM)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMCMD)
```

INSTRUCTION keyword example

If you wanted to add an XML processing instruction as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/css" href="mystyles.css"?>
```

then you could use the following logic:

```
CHANGE FIELD(#JSMCMD) TO("SET INSTRUCTION(xml-
stylesheet) DATA(type="text/css" href="mystyles.css")")
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMCMD)
```

RDML

* Define the fields used by the JSM Commands

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
Define Field(#JSMMSG) Type(*CHAR) Length(256)
Define Field(#JSMCMD) Type(*CHAR) Length(256)
Define Field(#JSMHND) Type(*CHAR) Length(4)
```

```
Define FIELD(#LINENUM) TYPE(*DEC) LENGTH(003) DECIMALS(0) LABEL('Line Number')
Define FIELD(#PARTNUM) TYPE(*DEC) LENGTH(003) DECIMALS(0) LABEL('Part Number')
Define FIELD(#PARTDSC) TYPE(*CHAR) LENGTH(020) LABEL('Part Description')
Define FIELD(#PARTAMT) TYPE(*DEC) LENGTH(010) DECIMALS(2) LABEL('Part Amount')
Define FIELD(#PARTQTY) TYPE(*DEC) LENGTH(003) DECIMALS(0) LABEL('Part Quantity')
Define Field(#LINSTAT) Type(*CHAR) Length(20) Label('Line Status')
Def_List Name(#RSPLINES) Fields(#LINENUM #PARTNUM #PARTDSC #PARTAMT #PARTQTY #LINSTAT)
```

* Set the customer details

```
#JSMCMD := 'SET FRAGMENT(CUSTOMER) SERVICE_EXCHANGE(*FRAGMENT)
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

* Set the order details by using SET FRAGMENT a number of times
Selectlist Named(#RSPLINES)

```
#JSMCMD := 'SET FRAGMENT(LINE) SERVICE_EXCHANGE(*FIELD)'  
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get  
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
```

```
#JSMCMD := 'SET FRAGMENT(PART) SERVICE_EXCHANGE(*FIELD)'  
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get  
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)  
Endselect
```

DELETE

This keyword can be used to delete a file from a specified directory.

```
DELETE ----- FILE ----- file path -----|
```

Keywords

FILE This keyword is used to define the file name to be deleted. The file path must be included.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

- Absolute path.
For example:
/orders/order.xml
C:/orders/order.xml
C:\orders\order.xml

or

- Relative path.
For example, orders/order.xml (note, no '/' at the start), in which case the document order.xml will be deleted from the orders directory under the JSM Instance directory on your server.

For the IBM i you can specify:

- Absolute path.
For example:
/orders/order.xml

or

- Relative path.
For example, orders/order.xml (note, no '/' at the start), in which case the document order.xml will be deleted from the orders directory under the JSM Instance directory on your server.

This keyword is mandatory.

Examples

RDML

* Define JSM fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Delete

```
CHANGE FIELD(#JSMCMD) TO("DELETE FILE(orders/order.xml)")
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

* Define the fields used by the JSM Commands

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
Define Field(#JSMMSG) Type(*CHAR) Length(256)
Define Field(#JSMCMD) Type(*CHAR) Length(256)
Define Field(#JSMHND) Type(*CHAR) Length(4)
```

* Delete

```
Change Field(#JSMCMD) To("DELETE FILE(orders/order.xml)")
Use Builtin(JSMX_COMMAND) With_Args(#JSMCMD) To_Get(#JSMSTS :
```

RENAME

The RENAME command can be used to rename a specified local file to a new name.

The file specified in for the TO value must be in the same directory as the file specified in the FROM value.

Required

```
RENAME ----- FROM ----- file path ----->
```

```
>-- TO ----- file path ----->
```

Optional

```
>-- REPLACE ----- *NO -----|  
      *YES
```

Keywords

FROM This keyword specifies the name and path of the file whose name is to be changed.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

- Absolute path.
For example:
/orders/order.xml
C:/orders/order.xml
C:\orders\order.xml

or

- Relative path.
For example, orders/order.xml (note, no '/' at the start), in which case the document order.xml must be a file located in

the orders directory under the JSM Instance directory on your server.

For the IBM i you can specify:

- Absolute path.
For example:
/orders/order.xml

or

- Relative path.
For example, orders/order.xml (note, no '/' at the start), in which case the document order.xml must be a file located in the orders directory under the JSM Instance directory on your server.

This keyword is mandatory.

TO This keyword specifies the name and the path to which the file is to be changed to. The path must be the same as that specified in the FROM keyword.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

Absolute path.

For example:

/orders/order2.xml

C:/orders/order2.xml

C:\orders\order2.xml

or

Relative path.

For example, orders/order2.xml (note, no '/' at the start), in which case the document order2.xml will be placed in the orders directory under the JSM Instance directory on your server.

For the IBM i you can specify:

Absolute path.

For example:
/orders/order2.xml

or

Relative path.

For example, orders/order2.xml (note, no '/' at the start), in which case the document order2.xml will be placed in the orders directory under the JSM Instance directory on your server.

This keyword is mandatory.

REPLACE This keyword is used to specify whether you want to overwrite any existing file of the same name as specified in the TO keyword.

There are two possible values:

***YES** - specifying this value will overwrite any existing file of the same name in the specified directory..

***NO** - if this value is set then an exception will occur if a file of the same name already exists in the directory. If you capture this exception you can report it back to the user.

The default value is *NO.

This keyword is optional.

Comments / Warnings

If you wish to place the file into another directory, you might want to consider using the ARCHIVE keyword on the READ command. This keyword will move the file that you are currently READING and place it into a new specified directory with whatever name you give it.

Examples

RDML

* Define JSM fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Rename

```
CHANGE FIELD(#JSMCMD) TO("RENAME FROM(/orders/test.xml) TO(/orders/test.xml)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMCMD)
```

RDMLX

* Define the fields used by the JSM Commands

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
```

```
Define Field(#JSMMSG) Type(*CHAR) Length(256)
```

```
Define Field(#JSMCMD) Type(*CHAR) Length(256)
```

```
Define Field(#JSMHND) Type(*CHAR) Length(4)
```

* Rename

```
Change Field(#JSMCMD) To("RENAME FROM(/orders/test.xml) TO(/orders/test.xml)
Use Builtin(JSMX_COMMAND) With_Args(#JSMCMD) To_Get(#JSMSTS ;
```

LIST

The LIST command will return a list of file names found within a specified directory.

This command may be useful if a number of XML documents need to be processed. The command could be used to populate a working list with all the documents that have been placed in a specific directory, then place the READ, BIND, and GET commands with a SELECT_LIST working on this list.

```
LIST ----- DIR ----- directory path ----->
>-- EXT ----- file extension ----->
>-- SORT ----- *NONE ----->
      *NAME
      *MODIFIED
>-- REVERSE ----- *YES -----|
      *NO
```

Keywords

- DIR This keyword is used to specify the directory that is to be searched.
- It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.
- The format of the path should be as follows:
- For Windows you can specify:
- Absolute path.
For example:
/orders
C:/orders
C:\orders
- or
- Relative path.

For example, `orders` (note, no `'` at the start), in which case the search will be conducted in the `orders` directory under the JSM Instance directory on your server.

For the IBM i you can specify:

- Absolute path.

For example:

`/orders`

or

- Relative path.

For example, `orders` (note, no `'` at the start), in which case the search will be conducted in the `orders` directory under the JSM Instance directory on your server.

This keyword is optional.

EXT This keyword is used to filter the list returned based on the file extension.

The filtering match is case insensitive.

This keyword is optional.

SORT The optional sort keyword allows sorting on file name or modified date.

The default value is `*NONE`.

REVERSE The optional reverse keyword allows the sorted order to be reversed.

The default value is `*NO`.

Comments / Warnings

If you do not specify the `DIR` value, then it will automatically return a list of files from the JSM instance directory.

Lists and Variables

This keyword requires a single field working list to receive the canonical file paths. To enable this, you should do the following:

1. Define a single field working list that will hold the returned list of files. The field needs to be long enough to hold the full canonical path and file name.

2. Use the SERVICE_LIST keyword with the LIST command. The service list value should include the name of the single field defined in the working list without the '#'.
3. In the TO_GET portion of the JSM_COMMAND Built-In Function, include the name of the working list defined above.

Examples

RDML

* Define the fields used by the JSM Commands

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
```

```
Define Field(#JSMMSG) Type(*CHAR) Length(256)
```

```
Define Field(#JSMCMD) Type(*CHAR) Length(256)
```

* Define field to hold file names

```
DEFINE FIELD(#FILENAME) TYPE(*CHAR) LENGTH(250)
```

```
DEF_LIST NAME(#FILELSTW) FIELDS(#FILENAME) COUNTER(#LIST
```

```
#JSMCMD := 'LIST DIR(NEWORDERS) SERVICE_LIST(FILENAME) EX'  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

* Define the fields used by the JSM Commands

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
```

```
Define Field(#JSMMSG) Type(*CHAR) Length(256)
```

```
Define Field(#JSMCMD) Type(*CHAR) Length(256)
```

```
Define Field(#JSMHND) Type(*CHAR) Length(4)
```

* Define field to hold file names

```
Define Field(#FILENAME) Type(*CHAR) Length(250)
```

```
Def_List Name(#FILELSTW) Fields(#FILENAME) Counter(#LISTCOUNT)
```

```
#JSMCMD := 'LIST DIR(NEWORDERS) SERVICE_LIST(FILENAME) EX'  
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

CLOSE

This command is used to close the current bind.

```
CLOSE ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Comments / Warnings

If you are planning to work with more than one XML document in a single function, then it is recommended that you CLOSE the bind of each document before you BIND the next XML document. This is not mandatory but it will free up resources.

Examples

RDML

```
* Define the fields used by the JSM Commands  
Define Field(#JSMSTS) Type(*CHAR) Length(020)  
Define Field(#JSMMSG) Type(*CHAR) Length(256)  
  
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

RDMLX

```
* Define the fields used by the JSM Commands  
Define Field(#JSMSTS) Type(*CHAR) Length(020)  
Define Field(#JSMMSG) Type(*CHAR) Length(256)  
Define Field(#JSMHND) Type(*CHAR) Length(4)  
  
Use Builtin(JSMX_CLOSE) With_Args(#JSMHND) To_Get(#JSMSTS #JSM
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSA Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
                *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE *SERVICE - read service trace file.
*TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.messageType (an application-defined property) in the service properties file. This property might

be defined in the service properties file with a line like this (the value "html" will be returned to the application in this case):

```
com.acme.property.messageType=html
```

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service and removes the temporary directory.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 service_unload) to_get(#jsi
```

5.17.5 XMLBindFileService Examples

Examples are supplied for the most common client programming languages:

[RDML](#)

[RDMLX](#)

5.18 XMLParserService

Service Name: **XMLParserService**

The XMLParserService allows XML documents to be read using different transport protocols.

The XMLParserService supports the following commands:

5.18.1 [SERVICE_LOAD](#)

5.18.2 [SERVICE_GET](#)

5.18.3 [SET](#)

5.18.4 [RESET](#)

5.18.5 [LOAD](#)

5.18.6 [PARSE](#)

5.18.7 [STORE](#)

5.18.8 [TRANSFORM](#)

5.18.9 [GET](#)

5.18.10 [CHECK](#)

5.18.11 [FOREACH](#)

5.18.12 [NEXT](#)

5.18.13 [SERVICE_UNLOAD](#)

5.18.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	SCHEMA		Optional. See SCHEMA .
	VALIDATING		Optional. See VALIDATING .

5.18.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.18.3 SET

The SET command allows the current node to be set to the specified path.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	NODE	*ROOT	Mandatory. Root path.
		value	Path name.

The SET command also allows an existing node value or node attribute value to be changed.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	NODE	*ROOT	Mandatory. Root path.
		*CURRENT	Current node.
		value	Path name.
	ATTRIBUTE	value	Optional. Name of attribute.
VALUE	value	Mandatory. Node or attribute value.	

5.18.4 RESET

The RESET command has no keywords and is used to reset the service to its initial state.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RESET			Reset the service to its initial state.

5.18.5 LOAD

LOAD command only has one mandatory keyword METHOD. The value of this keyword determines which content loader will be used to load the remote XML document. Other keywords on the LOAD command are passed to the content loader.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
LOAD	METHOD	*FILE	Mandatory. Load XML from local file system.
		*POP3	Load XML from POP3 server.
		*HTTP	Load XML from remote HTTP server.
		*HTTPS	Load XML from remote HTTP server using a secured connection.
		*FTP	Load XML from remote FTP server.
		*STORAGE	Load XML from internal storage.
		*BYTEARRAY	Load XML from command byte array.
	ENCODING	*DEFAULT	Optional. JVM default encoding.
		*CLIENT	Client encoding.
		value	Default. Auto detect encoding.
	FILE	value	Conditional. FTP file name or local file name.

HOST	host:port	Conditional. FTP, POP3 or HTTP host. The host name is specified as an IP address nnn.nnn.nnn.nnn:port or as a domain name.
USER	value	Conditional. HTTP, FTP or POP3 user profile.
PASSWORD	value	Conditional. HTTP, FTP or POP3 password for user profile specified in the USER keyword.
DATALINK	*PORT	Conditional. Used with METHOD(*FTP). FTP port mode.
	*PASV	Conditional. Default. Use with METHOD(*FTP). FTP passive mode.
MODE	*ASCII	Conditional. Used with METHOD(*FTP). Sets the FTP mode to ASCII.
	*BINARY	Used with METHOD(*FTP). Sets the FTP mode to BINARY.
NAMEFMT	0 or 1	Conditional. Used with METHOD(*FTP). Sets the FTP NAMEFMT mode.
PROXY	value	Conditional. HTTP proxy server.
PROXYUSER	value	Conditional. HTTP proxy user.

PROXYPASSWORD	value	Conditional. HTTP proxy password.
SUBJECT	value	Conditional. Used with METHOD(*POP3). Use mail message matching this subject.
FROM	value	Conditional. Used with METHOD(*POP3). Use mail message matching from address.
ATTACHMENT	value	Conditional. Used with METHOD(*POP3). Mail attachment name.
NAME	*DEFAULT	Conditional. Default. Use with the METHOD(*STORAGE)
	value	
IGNORE-PREFIX	*YES	Ignore namespace prefix.
	*NO	Default.

METHODS

***BYTEARRAY** source uses the byte array object from the command object.

ENCODING (*DEFAULT | *CLIENT | value) – optional default to auto detect encoding.

***FILE** source reads the specified file.

ENCODING (*DEFAULT | *CLIENT | value) – optional default to auto detect encoding.

FILE (filename)

***FTP** source reads the specified file from the remote host.

ENCODING (*DEFAULT | *CLIENT | value) – optional default to auto detect

encoding.

FILE (filename)

HOST (host:port) – port defaults to 21

USER (name)

PASSWORD (password)

DATALINK (*PASV | *PORT) – optional defaults to *PASV

MODE (*ASCII | *BINARY) – optional defaults to *BINARY

***POP3** source reads the first attachment from the first mail entry from the specified post office.

ENCODING (*DEFAULT | *CLIENT | value) – optional default to auto detect encoding.

HOST (smtpserver) - mandatory

USER (user) - mandatory

PASSWORD (password) - mandatory

FROM (from) - optional

SUBJECT (subject) - optional

ATTACHMENT (attachment) - optional

If no mail is found in the post office, then the returned response status is NOMAIL. If no useable mail entries are found, then the returned response status is NOATTACHMENT.

If no FROM, SUBJECT or ATTACHMENT keywords are present, then the first mail entry with an attachment is used. This mail entry is then deleted from the post office and contents of the attachment become the loaded source.

To add extra selection criteria to which mail entry is selected, use the FROM, SUBJECT or ATTACHMENT keywords singularly or together to form a logic AND condition.

***HTTP** and ***HTTPS** source reads the specified file from the remote host.

ENCODING (*DEFAULT | *CLIENT | value) – optional default to auto detect encoding .

FILE (filename) - mandatory

HOST (host:port) - optional defaults to LOCALHOST

USER (name) - optional

PASSWORD (password) - optional

PROXY (host:port) - optional

PROXYUSER (name) - optional

PROXYPASSWORD (password) - optional

*STORAGE source reads the source from the specified storage name.

NAME (*DEFAULT | name) - optional

5.18.6 PARSE

Two XML sources exist in the XMLParserService, the loaded source and the transformed source. Only one DOM object is active for traversal using GET, CHECK, FOREACH and NEXT commands.

The PARSE command parses the XML source and creates a DOM object. If no SOURCE keyword is present, the default value will be *LOAD.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
PARSE	SOURCE	*LOAD	Optional. Default. Loaded source is used.
		*TRANSFORM	Transformed source is used.

5.18.7 STORE

The STORE command allows the loaded or transformed source to be stored to an internal storage area for later retrieval using the LOAD METHOD (*STORAGE) NAME (name) command, or to be written to an external file. When saving to an external file the UTF-8 encoding is used.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
STORE	FILE	value	Optional. Local file name.
	NAME	*DEFAULT	Optional. Default. Storage name.
		value	
	OBJECT	*LOAD	Optional. Default. Store XML source.
		*TRANSFORM	Store XML transformed source.
*DOCUMENT		Store active DOM object to specified file.	
ENCODING		Optional. See ENCODING . Default encoding is UTF-8.	

If no object keyword is present the load source is stored or saved to file.

5.18.8 TRANSFORM

TRANSFORM command is used to transform the loaded XML source into another flavor of XML, this resultant XML is stored as the transform source. The program still has access to the original load source as well as the new transform source.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
TRANSFORM	XSL	value	Mandatory. See XSL .

If an optional list object argument was used with the TRANSFORM command, the XSL should produce Function XML and the data from the result XML will bind to the client program and no resultant transform source XML will be created.

Example

```
TRANSFORM XSL (name)
```

5.18.9 GET

The GET command returns data from the active DOM tree.

Except for the GET OBJECT (*MESSAGES) command all other values are return via the response message field. To receive XML parser messages a list object argument needs to be used with the command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	OBJECT	*MESSAGES	Optional. Get XML parser validation messages.
		*ROOTTAGNAME	Get XML root tag name.
	NODE	*ROOT	Get optional value of root node.
		*CURRENT	Get value of current node.
	ATTRIBUTE	value	Get value of specified node.
		value	Optional but requires NODE keyword.

5.18.10 CHECK

The CHECK command checks for the existence of a node or attribute. Also can test if a node or attribute has a particular value.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CHECK	NODE	*ROOT	Mandatory.
		*CURRENT	
		value	
	ATTRIBUTE	value	Optional.
	VALUE	value	Optional.

If the node does not exist in the DOM tree, then a response status of NOT_EXIST will be returned

If no VALUE or ATTRIBUTE keyword are present a response status of EXIST will be returned.

If no ATTRIBUTE keyword is present and a VALUE keyword is present, a case insensitive comparison is done between the node value and the VALUE keyword value. If they are equal then a response status of EQUAL is returned, else a NOT_EQUAL is returned.

If no VALUE keyword is present and ATTRIBUTE keyword is present, then the existence of the attribute is done and an EXIST or NOT_EXIST response status is returned.

If both a VALUE and ATTRIBUTE keyword are present a value comparison of the attribute value is done and an EQUAL or NOT_EQUAL response status is returned.

5.18.11 FOREACH

The FOREACH command creates a list of nodes specified by the NODE path value. If no NODELIST keyword is present a default name of *DEFAULT is used.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
FOREACH	NODE	value	Mandatory.
	NODELIST	*DEFAULT	Optional. Default. Name of node list to receive selected nodes.
		value	

5.18.12 NEXT

The NEXT command sets the current node to the next node in the specified node list.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
NEXT	OBJECT	*NODE	Mandatory.
	NODELIST	*DEFAULT value	Optional. Default.

If no more nodes exist in the specified list, then the response status returned is NONE.

5.18.13 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.18.14 XMLParserService Examples

For RDML code examples, go to [XMLParserService 1](#) and [XMLParserService 2 \(Node traversal\)](#).

5.19 XMLReaderService

The XMLReaderService allows XML files to be read using a StAX stream reader.

The Streaming API for XML (StAX) is an API for pull-parsing XML.

The streaming API gives parsing control to the programmer by exposing a simple iterator based API.

This allows the programmer to ask for the next event (pull the event) and allows state to be stored in a procedural fashion.

The XMLReaderService uses the Woodstox API (Refer to <http://woodstox.codehaus.org/>).

5.19.1 What can I use the XMLReaderService for?

If you need to determine the root element of a large XML file, then the XMLReaderService allows the first element to be read and the parsing activity to be ended.

5.19.2 Using the XMLReaderService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application that reads an XML file would typically issue the following sequence of commands:

JSM(X)_OPEN**JSM(X)_COMMANDs**

SERVICE_LOAD

SET

OPEN

NEXT

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Refer to [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.19.3 XMLReaderService Commands

Your application issues commands to the XMLReaderService by passing the command strings through the Java Service Manager using the JSM_COMMAND or JSMX_COMMAND Built-In Function or an API for your chosen development language.

The commands that the XMLReaderService processes are:

SERVICE_LOAD

SET

OPEN

NEXT

CLOSE

SERVICE_GET

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
--->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name ----->  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

```
>-- VALIDATING ----- *NO -----|  
      *YES
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

SERVICE The name of the service to be loaded - in this case XMLReaderService.

TRACE To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.

The possible values for the TRACE keyword are:

- *NO

- *YES
- *ERROR

The trace option *ERROR will turn on tracing and if the service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.

TRACE_NAME The optional TRACE_NAME keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

VALIDATING Optional. See [VALIDATING](#).

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(XMLReader
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jsmcommand := 'service_load service(XMLReaderService)'
use builtin(jsmx_command) with_args(#jsmxhdl1 #jsmcommand) to_get(#jsn
```

SET

The SET command sets the current directory.

Required

```
SET ---- DIR ----- directory path -----|
```

Keywords

DIR This keyword is used to specify the default directory.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET DIR(/orders)') TO_G:
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'set dir(/orders)') to_get(#js
```

OPEN

The OPEN command opens an existing XML file.

Required

OPEN ---- *FILE* ----- *file path* -----|

Keywords

FILE This keyword is used to specify the file name and path of the xml file.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

- Absolute path
For example:
/orders/order.xml
C:/orders/order.xml
C:\orders\order.xml

or

- Relative path.
For example, orders/order.xml (note, no '/' at the start), in which case the order.xml file must reside in the orders directory under the JSM Instance directory on your server.

For the IBM i you can specify:

- Absolute path.
For example:
/orders/order.xml

or

- Relative path.
For example, orders/order.xml (note, no '/' at the start), in which case the order.xml file must reside in the orders directory under the

JSM Instance directory on your server.

Note: Whatever directory structure you specify must already exist.
This keyword is mandatory.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('OPEN FILE(ORDER.XM
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'open file(order.xml)') to_g
```

NEXT

The NEXT command reads the next streaming event.

Each event will return a status code to identify the type of event received.

When all events have been received a status code of NONEXT is returned to identify the end of the stream.

All events do not need to be read before closing the stream reader.

Element attribute names and values can be accessed by using a two field working list argument. The working list is cleared and attributes are added when a ELEMENTSTART event is received.

Possible status and message values

Status	Message
NONEXT	
ELEMENTSTART	Qualified element name. For example: {http://www.cars.com/xml}part
ELEMENTEND	Qualified element name. For example: {http://www.cars.com/xml}part
COMMENT	Comment text.
INSTRUCTION	Processing Instruction.

Required

NEXT

Optional

>-- COMMENT ----- *NO ----->
 *YES

```
>-- INSTRUCTION ----- *NO -----|  
      *YES
```

Keywords

COMMENT This optional keyword is used to include comment events.
The possible values for the COMMENT keyword are:

- *NO
- *YES

The default value is *NO.

INSTRUCTION This optional keyword is used to include processing instruction events.

The possible values for the INSTRUCTION keyword are:

- *NO
- *YES

The default value is *NO.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('NEXT SERVICE_LIST(A
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'next') to_get(#jsmxsts #jsn
```

CLOSE

The CLOSE command closes the current reader.

```
CLOSE ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(CLOSE) TO_GET(#JSMS'
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 close) to_get(#jsmxsts #jsn
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSa Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
        *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE *SERVICE - read service trace file.
*TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.messageType (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html"

will be returned to the application in this case):

```
com.acme.property.messageType=html
```

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service, closing any input or output streams and removing temporary directories or files.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TC
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 service_unload) to_get(#jsi
```

5.20 XMLWriterService

The XMLWriterService allows XML files to be created using a StAX stream writer.

The Streaming API for XML (StAX) is an API for stream writing XML.

The XMLWriterService uses the Woodstox API (Refer to <http://woodstox.codehaus.org/>).

5.20.1 What can I use the XMLWriterService for?

If you need to create a large XML file, the XMLWriterService allows XML elements to be streamed out to the file as they are being created.

Other approaches to creating XML files, require the whole XML content to be created in-memory before outputting to a file.

5.20.2 Using the XMLWriterService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application that creates an XML file would typically issue the following sequence of commands:

JSM(X)_OPEN**JSM(X)_COMMANDs**

SERVICE_LOAD

SET

OPEN

WRITE

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Refer to [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.20.3 XMLWriterService Commands

Your application issues commands to the XMLWriterService by passing the command strings through the Java Service Manager using the JSM_COMMAND or JSMX_COMMAND Built-In Function or an API for your chosen development language.

The commands that the XMLWriterService processes are:

SERVICE_LOAD

SET

OPEN

WRITE

CLOSE

SERVICE_GET

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
--->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

SERVICE The name of the service to be loaded - in this case XMLWriterService.

TRACE To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.

The possible values for the TRACE keyword are:

- *NO
- *YES
- *ERROR

The trace option *ERROR will turn on tracing and if the service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.

TRACE_NAME The optional TRACE_NAME keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(XMLWriterService)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMCMD)
```

RDMLX

```
#jxmcommand := 'service_load service(XMLWriterService)'
use builtin(jsmx_command) with_args(#jxmhandle1 #jxmcommand) to_get(#jxmcommand)
```

SET

The SET command can be used to set the current directory. This command also allows XML prefix to namespace mappings used during the XML creation process to be declared at the root scope or with element start and element end scope.

Optional

```
SET ---- DIR ----- directory path ----->
>-- OBJECT ----- *PREFIX ----->
      *DEFAULTNS
>-- PREFIX ----- prefix ----->
>-- NAMESPACE ---- namespace -----|
```

Keywords

- | | |
|-----------|--|
| DIR | This keyword is used to specify the default directory. |
| OBJECT | This keyword is used to define an XML namespace. Depending on its usage, it can either be used to define a namespace in the root scope or within an element start and element end scope.
The possible values for the OBJECT keyword are: <ul style="list-style-type: none">• *PREFIX• *DEFAULTNS |
| PREFIX | This keyword specifies an XML prefix and is only used with the OBJECT type of *PREFIX. |
| NAMESPACE | This keyword specifies an XML namespace and is used with the OBJECT type *PREFIX or *DEFAULTNS. |

Example

```
SET OBJECT ( *DEFAULTNS ) NAMESPACE ( http://parts.com )
```

```
SET OBJECT ( *PREFIX ) PREFIX ( abc ) NAMESPACE ( http://salesorder.c
```

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET DIR(/orders)') TO_G.
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 'set dir(/orders)') to_get(#js
```

OPEN

The OPEN command creates a new XML file or replaces an existing file.

Required

OPEN ---- *FILE* ----- *file path* -----|

Keywords

FILE This keyword is used to specify the file name and path of the xml file.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

- Absolute path
For example:
/orders/order.xml
C:/orders/order.xml
C:\orders\order.xml

or

- Relative path.
For example, orders/order.xml (note, no '/' at the start), in which case the order.xml file must reside in the orders directory under the JSM Instance directory on your server.

For the IBM i you can specify:

- Absolute path.
For example:
/orders/order.xml

or

- Relative path.
For example, orders/order.xml (note, no '/' at the start), in which case the order.xml file must reside in the orders directory under the

JSM Instance directory on your server.

Note: Whatever directory structure you specify must already exist.
This keyword is mandatory.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('OPEN FILE(ORDER.XM
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'open file(order.xml)') to_g
```

WRITE

The WRITE command is used to create and write out XML components. The OBJECT keyword specifies the type of XML component to create.

Required

```
WRITE ---- OBJECT ----- *DOCUMENTSTART -----  
----->  
    *DOCTYPE  
    *COMMENT  
    *INSTRUCTION  
    *NAMESPACE  
    *ELEMENTSTART  
    *TEXT  
    *CDATA  
    *ELEMENTEND  
    *DOCUMENTEND
```

Optional

```
>-- NAME ----- qualified name ----->  
>-- PREFIX ----- prefix ----->  
>-- NAMESPACE ----- namespace ----->  
>-- TEXT ----- text ----->  
>-- TARGET ----- target ----->  
>-- DATA ----- data -----|
```

Keywords

OBJECT This mandatory keywords specifies the type of XML element to be created.

The possible values for the OBJECT keyword are:

- *DOCUMENTSTART
- *DOCTYPE
- *COMMENT
- *INSTRUCTION
- *NAMESPACE
- *ELEMENTSTART
- *TEXT
- *CDATA
- *ELEMENTEND
- *ELEMENTEMPTY
- *DOCUMENTEND

NAME	<p>This keyword specifies the qualified XML element name. This keyword is used with the following OBJECT types.</p> <ul style="list-style-type: none">● *ELEMENTSTART● *ELEMENTEMPTY <p>The notation for expressing a qualified element name as a string is to enclosed the namespace URI inside curly brackets and prefix the element name with this namespace. For example: {http://www.cars.com/xml}part</p> <p>Element attributes can be created by using a two field working list that contains qualified attribute name and values as an argument to the WRITE OBJECT(*ELEMENTSTART) command.</p>
PREFIX	<p>This keyword specifies an XML prefix and is only used with the OBJECT type of *NAMESPACE.</p>
NAMESPACE	<p>This keyword specifies an XML namespace and is only used with the OBJECT type of *NAMESPACE.</p>
TEXT	<p>This keyword specifies the text value. This keyword is used with the following OBJECT types.</p> <ul style="list-style-type: none">● *DOCTYPE● *COMMENT

- *TEXT
- *CDATA

TARGET This keyword specifies the processing instruction target and is only used with the OBJECT type of *INSTRUCTION.

DATA This keyword specifies the processing instruction data and is only used with the OBJECT type of *INSTRUCTION.

Example

OPEN FILE (order.xml)

WRITE OBJECT (*DOCUMENTSTART)

WRITE OBJECT (*DOCTYPE) TEXT (<!DOCTYPE Orders SYSTEM "or

WRITE OBJECT (*COMMENT) TEXT (Some comment text)

WRITE OBJECT (*INSTRUCTION) TARGET (action) DATA (reply)

SET OBJECT (*PREFIX) PREFIX (abc) NAMESPACE (http://salesorder.c

SET OBJECT (*DEFAULTNS) NAMESPACE (http://parts.com)

WRITE OBJECT (*ELEMENTSTART) NAME ({http://acme.com}Orders)

WRITE OBJECT (*TEXT) TEXT (Some text)

WRITE OBJECT (*ELEMENTEND)

WRITE OBJECT (*ELEMENTEMPTY) NAME ({http://acmme.com}Sales

WRITE OBJECT (*ELEMENTSTART) NAME ({http://acme.com}Address

WRITE OBJECT (*CDATA) TEXT (Some text)

WRITE OBJECT (*NAMESPACE) PREFIX (def) NAMESPACE (http://ac

```
WRITE OBJECT ( *ELEMENTEND )
```

```
WRITE OBJECT ( *DOCUMENTEND )
```

```
CLOSE
```

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('WRITE OBJECT(*DOCU
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'write object(*documentsta
```

CLOSE

The CLOSE command closes the current reader.

```
CLOSE ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(CLOSE) TO_GET(#JSMS'
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 close) to_get(#jsmxsts #jsn
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSA Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
        *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE *SERVICE - read service trace file.
*TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.message type (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html")

will be returned to the application in this case):

```
com.acme.property.messageType=html
```

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service, closing any input or output streams and removing temporary directories or files.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TC
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 service_unload) to_get(#jsi
```

5.21 XMLQueryService

The XMLQueryService permits an application to selectively interrogate values contained in an XML document using XML Path Language (XPath) expressions.

It is not suitable or intended for and usually would not be used to process the entire contents of an XML document. There are other LANSAs Integrator services (mentioned below) that are much more appropriate for that.

Rather it is intended for limited and selective interrogation of particular values from the XML document. In some applications, for example, it may be necessary for the application to determine certain key values from the XML document in order to decide how to proceed with further processing or for use in the course of further processing.

NOTE: for LANSAs Composer users, the supplied XML_QUERY activity provides this functionality.

NOTE: The XMLQueryService loads the entire XML document into memory when processing your queries. Application performance can degrade when used with exceptionally large XML files.

Related Services

The XMLQueryService is not dependent on other services.

As noted above, the service is not suitable or intended for processing the entire contents of an XML document. LANSAs Integrator provides a number of other XML services that may be more suitable for such purposes including:

- XMLBindFileService
- XMLReaderService
- XMLWriterService

Technical Specifications

The service is implemented using features of the javax.xml.xpath package. The following XML standards apply:

- [XML Path Language \(XPath\)](#)
- LANSAs Integrator must be using a Java 5 or above JRE in order to use this service.

5.21.1 What can I use the XMLQueryService for?

Use the XMLQueryService when you need to selectively interrogate a limited number of particular values in an XML document.

For example, suppose you have an application that receives and processes sales orders in an agreed XML format.

One part of your application might process the sales order XML document in its entirety, perhaps using other LANSA Integrator services or a LANSA Composer Transformation Map to read the contents and update your application database.

However, an independent code unit in your application is required to email an acknowledgement of the order. In order to do so, the most convenient means to access the customer's return email address is directly from the corresponding element value in the original sales order XML document.

In such circumstances, your application could use the XMLQueryService to selectively and efficiently address and retrieve just the value of the element (or attribute) in the sales order XML document that contains the customer's return email address.

5.21.2 Using the XMLQueryService

Refer to the following for general information on XMLQueryService usage:

- [Typical XMLQueryService Command Usage](#)
- [Quick Guide to XPath expressions for use with XMLQueryService](#)

Typical XMLQueryService Command Usage

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG or C, you have to complete the same basic steps.

For example, an application that interrogates values from an XML document using the XMLQueryService would typically issue the following sequence of commands:

JSM(X)_OPEN

JSM(X)_COMMANDs

SERVICE_LOAD

LOAD

QUERY

SERVICE_UNLOAD

JSM(X)_CLOSE

Refer to Java Service Manager Clients for the command details that apply to your chosen development language.

Quick Guide to XPath expressions for use with XMLQueryService

There are many XPath resources available on the web. To get you started, you could try the following:

- For one quick and easy introduction to XPath: [XPath Tutorial](#)
- [XPath Examples](#)
- [XML in a Nutshell – A Desktop Quick Reference](#)
- [XML Path Language \(XPath\) Version 1.0](#)

This document does not intend or purport to provide a definitive description of or reference to XPath expression syntax. However, for those readers who have not used XPath expressions before, this section will give a brief overview and examples that might help you get started with the XMLQueryService. Refer to the following topics in this section:

- ExampleXML
- Introduction to XML Path Language (XPath)
- XPath Examples for use with XMLQueryService
- XML Namespaces and How They Affect XPath Expressions for XMLQueryService

Example XML

The examples provided later in this section will refer to the following simple example XML document:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE Orders SYSTEM "http://www.lansa.com/schemas/tutorder.dtd"

<Orders>

  <SalesOrder SONumber="12345">

    <Customer CustNumber="543">
      <CustName>ABC Industries</CustName>
      <Street>123 North St.</Street>
      <City>Bankstown</City>
      <State>NSW</State>
```

```
<PostCode>2087</PostCode>
</Customer>

<OrderDate>2012-11-19</OrderDate>

<Line LineNumber="1">
  <Part PartNumber="123">
    <Description>Gasket Paper</Description>
    <Price>9.95</Price>
  </Part>
  <Quantity>10</Quantity>
</Line>

<Line LineNumber="2">
  <Part PartNumber="456">
    <Description>Glue</Description>
    <Price>13.27</Price>
  </Part>
  <Quantity>5</Quantity>
</Line>

</SalesOrder>

</Orders>
```

Introduction to XML Path Language (XPath)

XPath is a syntax for constructing path expressions to select nodes in an XML document. To some extent, these path expressions look very similar to path expressions you use when working with the file system on your computer.

In general, XPath recognises seven types of nodes, viz. element, attribute, text, namespace, processing-instruction, comment, and document nodes. In the context of the XMLQueryService we are chiefly concerned with the element and attribute nodes and of course, the document node.

In the example XML document shown above, some of the *elements* are <Orders>, <SalesOrder>, <Customer>, <OrderDate>, <Line> and <Part>, while the *attributes* include SONumber=, CustNumber=, LineNumber= and PartNumber=.

The following is an example XPath expression that will select the PartNumber= attribute of the first <Part> element in the second <Line> element in the first <SalesOrder> element of the example XML document:

```
/Orders/SalesOrder[1]/Line[2]/Part[1]/@PartNumber
```

Note that the selection of the <SalesOrder>, <Line> and <Part> elements in the above example are by ordinal index. In particular, the selection of the <Line> element does NOT refer to the value of the LineNumber= attribute (although that is possible too, as you will see later).

XPath provides a large number of built-in functions that can manipulate and compare values in a variety of ways for more advanced usage. For example, the following expression uses the *contains* built-in function to select all <Part> elements (wherever they occur) whose <Description> element contains the string "Paper":

```
//Part[contains(Description, "Paper")]
```

In XPath, you select a node or set of nodes, by following a path or steps. Your XPath expression will often include one or of the following:

- nodename** Selects all nodes with the specified name.
- / Selects from the root node
- // Selects nodes in the document from the current node that match the selection no matter where they are
- Selects the current node
- .. Selects the parent of the current node

@nodename Selects attributes with the specified name

In XPath, a predicate is a sub-expression contained in square brackets that is used to select a specific node or a node that contains a specific value. The following are some examples of XPath expressions that use predicates:

```
/Orders/SalesOrder[1]      Selects the first <SalesOrder> element that is a child of the <Orders> element.
```

```
/Orders/SalesOrder[last()]    Selects the last <SalesOrder> element that is a child of the <Orders> element. (In the particular instance of the example XML document shown, there is only one <SalesOrder> element and so the
```

result will be the same.)

`//Part[Price<=10.00]` Selects <Part> elements, wherever they occur, whose <Price> element has a value less than or equal to 10.00.

There is much more to know about XPath expressions. If you would like more information, you could start by referring to some of the links provided above.

Important note: XML node names are case sensitive. Your XPath expressions must specify the correct case when specifying element and attribute names. For example, the expression `//salesorder` is NOT the same as `//SalesOrder`. When used with the example XML document shown above, the former expression will FAIL to select ANY nodes, while the latter will select all <SalesOrder> elements, wherever they occur in the document.

XPath Examples for use with XMLQueryService

The following examples use XPath expressions in the parameters of the QUERY command of the XMLQueryService to select values from the example XML document shown above.

1. This example will select nothing because XML and XPath are case-sensitive and the wrong case is used to select the <SalesOrder> elements:

```
QUERY NODES(//SALESORDER) NODESVALUE1(@SONumber)
```

2. These two examples use alternate implementations to select all <SalesOrder> elements, and return the sales order number for each. Functionally, they are equivalent (when used with the example XML document):

```
QUERY NODES(//SalesOrder/@SONumber)
```

```
QUERY NODES(//SalesOrder) NODESVALUE1(@SONumber)
```

3. Selects all <SalesOrder> elements, and returns the customer number for each:

```
QUERY NODES(//SalesOrder) NODESVALUE1(Customer/@CustNumber)
```

4. Selects <Customer> elements that have a value of '543' for their customer number and returns the sales order number of the parent <SalesOrder> element:

```
QUERY NODES(//Customer[@CustNumber="543"]) NODESVALUE1(..@S
```

5. Selects all <Part> elements for the <SalesOrder> element(s) with the order number specified and returns the part number and quantity for each:

```
QUERY NODES(//SalesOrder[@SONumber="12345"]/Line/Part)
  NODESVALUE1(@PartNumber)
  NODESVALUE2(..Quantity)
```

6. Selects all <Part> elements with a price greater than 2.99 and, for each, returns the order number, the part number, the price, the quantity and calculates and returns the extended value (price * quantity):

```
QUERY NODES(//Part[Price>2.99])
  NODESVALUE1(..../@SONumber)
  NODESVALUE2(@PartNumber)
  NODESVALUE3(Price)
  NODESVALUE4(..Quantity)
  NODESVALUE5(Price*..Quantity)
```

XML Namespaces and How They Affect XPath Expressions for XMLQueryService

The examples used so far operate on an XML document that contains no explicit namespace declarations and does not make use of namespace prefixes. This is the simplest case, but frequently does not reflect the real world.

Consider this minor alteration to the example XML document that specifies a *default* namespace for the XML document:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE Orders SYSTEM "http://www.lansa.com/schemas/tutorder.dtd"

<Orders xmlns="urn:schemas-lansa-com:tutorder.dtd">

  ... etc ...

</Orders>
```

Where a document makes use a single default namespace like this, the easiest approach to formulating XPath expressions for use with it is usually to disregard the namespace. Since only one namespace is used and there are no namespace prefixes present on the node names, you can usually use the same expressions as you would use with the earlier example. Each of the following queries work successfully with the example document that declares the default namespace,

providing the document is *not* loaded in namespace-aware mode:

```
QUERY NODES(//SalesOrder/@SONumber)
```

```
QUERY NODES(//SalesOrder) NODESVALUE1(Customer/@CustNumber)
```

```
QUERY NODES(//Customer[@CustNumber="543"])  
  NODESVALUE1(../@SONumber)
```

```
QUERY NODES(//SalesOrder[@SONumber="12345"]/Line/Part)  
  NODESVALUE1(@PartNumber)  
  NODESVALUE2(..Quantity)
```

```
QUERY NODES(//Part[Price>2.99])  
  NODESVALUE1(..../@SONumber)  
  NODESVALUE2(@PartNumber)  
  NODESVALUE3(Price)  
  NODESVALUE4(..Quantity)  
  NODESVALUE5(Price*../Quantity)
```

However, in XML documents that use more than one namespace and/or implement namespace prefixes, things can get a little more complicated. Consider the following alternate example XML document and contrast it to the earlier example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE Orders SYSTEM "http://www.lansa.com/schemas/tutorder.dtd"
```

```
<tut:Orders xmlns:tut="urn:schemas-lansa-com:tutorder.dtd">
```

```
  <tut:SalesOrder SONumber="12345">
```

```
    <tut:Customer CustNumber="543">
```

```
      <tut:CustName>ABC Industries</tut:CustName>
```

```
      <tut:Street>123 North St.</tut:Street>
```

```
      <tut:City>Bankstown</tut:City>
```

```
      <tut:State>NSW</tut:State>
```

```
      <tut:PostCode>2087</tut:PostCode>
```

```
    </tut:Customer>
```

```

<tut:OrderDate>2012-11-19</tut:OrderDate>

<tut:Line LineNumber="1">
  <tut:Part PartNumber="123">
    <tut:Description>Gasket Paper</tut:Description>
    <tut:Price>9.95</tut:Price>
  </tut:Part>
  <tut:Quantity>10</tut:Quantity>
</tut:Line>

<tut:Line LineNumber="2">
  <tut:Part PartNumber="456">
    <tut:Description>Glue</tut:Description>
    <tut:Price>13.27</tut:Price>
  </tut:Part>
  <tut:Quantity>5</tut:Quantity>
</tut:Line>

</tut:SalesOrder>

```

```
</tut:Orders>
```

This document contains a namespace declaration and uses the associated namespace prefix on the element names. The use of namespace features and especially of namespace prefixes can complicate the syntax of the XPath expressions necessary for a given query.

Again you should refer to the many resources available on the web concerning XML namespaces and how they affect XPath. One such reference is:

- [XML Namespaces and how they affect Xpath and XSLT](#)

The easiest approach to formulating XPath expressions for use with such an instance document is to disregard the namespace(s). If the document is loaded WITHOUT the namespace-aware option (the default mode), then you can use *nearly* the same expressions as you would use with the earlier example. Each of the following queries work successfully with the namespace prefixed version of the document as shown above (note that the namespace prefix is omitted entirely from the XPath expressions):

```
QUERY NODES(/Orders/SalesOrder/@SONumber)
```

```
QUERY NODES(/Orders/SalesOrder) NODESVALUE1(Customer/@CustNun
```

```
QUERY NODES(/Orders/SalesOrder/Customer[@CustNumber="543"])
  NODESVALUE1(../@SONumber)
```

```
QUERY NODES(/Orders/SalesOrder[@SONumber="12345"]/Line/Part)
  NODESVALUE1(@PartNumber)
  NODESVALUE2(..Quantity)
```

```
QUERY NODES(/Orders/SalesOrder/Line/Part[Price>2.99])
  NODESVALUE1(..../@SONumber)
  NODESVALUE2(@PartNumber)
  NODESVALUE3(Price)
  NODESVALUE4(..Quantity)
  NODESVALUE5(Price*../Quantity)
```

If, however, your document declares more than one namespace, and, especially where there would be a namespace collision without the use of the namespaces, it may be necessary to load the document in namespace-aware mode. This is done by specifying **YES* for the *NAMESPACEAWARE* keyword on the *LOAD* command of the *XMLQueryService*. For example:

```
LOAD FILE(salesorder.xml) NAMESPACEAWARE(*YES)
```

However, once the document is loaded in namespace-aware mode, the example queries shown up to this point will no longer function because now the namespace forms a part of the identification of nodes in the XML document.

There are a variety of ways to formulate your XPath expressions such that they will function in the way you require in namespace-aware mode and it is well beyond the scope of this document to attempt to cover all the options. However, here are a few examples that might help to get you started:

1. This example uses the *local-name* XPath built-in function to select nodes based on their *local name* (the node name WITHOUT the namespace prefix):

```
QUERY NODES(//*[local-name() = 'SalesOrder']) NODESVALUE1(@SONumber)
```

2. If multiple namespaces are used and 'SalesOrder' is ambiguous in this context, then you can extend the previous example to use the *namespace-uri* XPath built-in function:

```
QUERY NODES(//*[local-name() = 'SalesOrder' and namespace-uri() = 'urn:schemas-lansa-com:tutorder.dtd'])
```

NODESVALUE1(@SONumber)

3. Alternatively, if you know that all instances of the XML document will use the same namespace prefixes (which, you should understand, is NOT strictly necessary for them to be valid, even though it may commonly be the case in practice), then you can include the namespace prefixes in your XPath expressions (provided the document *is* loaded in namespace-aware mode):

QUERY NODES(//tut:SalesOrder) NODESVALUE1(@SONumber)

In summary, each of the following queries work successfully with the namespace prefixed version of the document as shown above, providing the document *is* loaded in namespace-aware mode AND providing the actual namespace prefix used in the XML document matches that assumed in the queries:

QUERY NODES(//tut:SalesOrder/@SONumber)

QUERY NODES(//tut:SalesOrder) NODESVALUE1(tut:Customer/@CustNum

QUERY NODES(//tut:Customer[@CustNumber="543"])
NODESVALUE1(../@SONumber)

QUERY NODES(//tut:SalesOrder[@SONumber="12345"]/tut:Line/tut:Part)
NODESVALUE1(@PartNumber)
NODESVALUE2(..:/tut:Quantity)

QUERY NODES(//tut:Part[tut:Price>2.99])
NODESVALUE1(..../@SONumber)
NODESVALUE2(@PartNumber)
NODESVALUE3(tut:Price)
NODESVALUE4(..:/tut:Quantity)
NODESVALUE5(tut:Price*../tut:Quantity)

5.21.3 XMLQueryService Commands

Your application issues commands to the XMLQueryService by passing the command strings through the Java Service Manager using the JSM_COMMAND or JSMX_COMMAND built-in function, or an equivalent built-in function or API for your chosen development language.

The commands that the XMLQueryService processes are:

SERVICE_LOAD

LOAD

SET

RESET

QUERY

SERVICE_GET

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICENAME ----- servicename -----  
----->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Mapping Service Name to Java Classes](#)

[Service Program Tracing from the Client](#)

Keywords

SERVICE

The name of the service to be loaded - in this case XMLQueryService.

TRACE

To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.

The possible values for the TRACE keyword are:

- *NO
- *YES
- *ERROR

The trace option *ERROR will turn on tracing and if the service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.

The TRACE keyword will override the settings in the manager.properties file.

TRACE_NAME This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory. Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML Example:

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(XMLQUER
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
#jsmcommand := 'service_load service(XMLQUERYSERVICE)'
use builtin(jsmx_command) with_args(#jsmhandle #jsmcommand) to_get(#jsn
```

ILE RPG Example:

```
c          eval   jsmcmd = 'service_load'
c          + ' service(XMLQUERYSERVICE)'
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
```

LOAD

The LOAD command is used to parse and load the XML document whose contents are to be queried.

The LOAD command must be executed successfully before using the SET or QUERY commands.

Required

```
LOAD ----- FILE ----- file path ----->
```

Optional

```
>-- VALIDATE ----- *NO ----->  
      *YES
```

```
>-- NAMESPACEAWARE ---- *NO -----|  
      *YES
```

Keywords

FILE

The path and file name for the XML document whose contents are to be queried.

VALIDATE

By default (and if you specify *NO for this keyword) the XML document is parsed WITHOUT validating it against any DTD or schema it may reference. In this mode, the XML document is required to be well-formed, but conformance to the DTD or schema is NOT checked. For many typical applications of the XMLQueryService, validation is an unnecessary overhead. However, if you require validation to be performed, you may specify *YES for this keyword.

NAMESPACEAWARE

By default (and if you specify *NO for this keyword) the XML document is parsed in a non-namespace-aware mode. For most cases, this simplifies the form of the XPath expressions

necessary to perform a given query.

In some more complex documents (and especially for documents in which more than one namespace is referenced) it may be necessary to specify *YES for this keyword in order to load the document in namespace-aware mode.

Note that the value specified (or assumed) for this keyword will affect the form of XPath expressions necessary to successfully perform a given query.

Examples

RDML Example:

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('LOAD FILE(salesorder.xml)')
```

RDMLX Example:

```
use builtin(jsmx_command) with_args(#jsmhdl 'load file(salesorder.xml)') to_
```

ILE RPG Example:

```
c          eval    jsmcmd = 'load'  
c          + ' file(salesorder.xml)'  
c          callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
```

SET

The SET command is used to set the "current" node in the presently loaded XML document. The "current" node is the node to which further queries (using the QUERY command) are applied.

When the document is loaded using the LOAD command, the current node is the document node. It is not mandatory to issue the SET command to change the current node. Providing the XPath expressions used in queries are formulated appropriately, they can be executed against the document node, and in many cases this will be all that is needed.

However, in more complex applications of the XMLQueryService, the SET command may be used, perhaps iteratively, to process specific sections of an XML document. In particular, the generated XPath expressions returned by the QUERY command when the special *XPATH or *XPATH_CONCISE values are specified may be used with the SET command to iteratively process subsets of the document contents.

Required

```
SET ----- CURRENTNODE ----- xpath expression -----  
---|  
                                     *DOCUMENT
```

Keywords

CURRENTNODE The value of this keyword is used to set the current node in the loaded XML document as the node to which further queries (using the QUERY command) are applied.

The special value *DOCUMENT specifies that the document node is the current node – this is the default state immediately after loading an XML document with the LOAD command.

Alternatively you may specify an appropriately formulated XPath expression that identifies a single node in the document that is to be the "current" node.

For more information about XPath expressions used with

the XMLQueryService, refer to [Quick Guide to XPath expressions for use with XMLQueryService](#).

Examples

The following examples use the SET command to set the current node to the FIRST instance of an element anywhere in the document with the element name 'SalesOrder'.

RDML Example:

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET CURRENTNODE(//S
```

RDMLX Example:

```
use builtin(jsmx_command) with_args(#jsmhdl 'set currentnode(//SalesOrder[
```

ILE RPG Example:

```
c          eval   jsmcmd = 'set'  
c          + ' currentnode(//SalesOrder[1])'  
c          callp  p_jsmcmd(jsmcmd;jsmsts:jsmmsg)
```

RESET

The RESET command is used to unload the current XML document and reset the state of the service ready for processing a new document. Any memory occupied by the currently loaded document will become eligible to be freed by the garbage collector in due course.

Required

RESET ----- no keywords -----|

Keywords

There are no keywords used with the RESET command

Examples

The following examples use the RESET command to unload the currently loaded XML document.

RDML Example:

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('RESET') TO_GET(#JSMS
```

RDMLX Example:

```
use builtin(jsmx_command) with_args(#jsmhdl 'reset') to_get(#jsmsts #jsmms
```

ILE RPG Example:

```
c          eval   jsmcmd = 'reset'  
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
```

QUERY

The QUERY command is used to selectively interrogate values contained in the presently loaded XML document using XPath expressions.

You must provide a working list with the QUERY command in which the result value(s) will be returned to your program. The working list will be filled with zero, one or more entries – one entry for each selected node (resulting from the query in the NODES keyword).

The working list must contain at least one field in which the value for the node specified by the NODESVALUE1 keyword will be placed.

It must contain one additional field for each of the NODESVALUE2 through NODESVALUE5 keywords for which values are specified. For example, if you specify all five keywords NODESVALUE1 through NODESVALUE5, then your working list must provide five fields in which to receive the resulting values for each selected node.

Required

```
QUERY ----- NODES ----- xpath expression -----  
->
```

Optional

```
>--- NODESVALUE1 ----- . ----->  
    *LOCALNAME  
    *NAMESPACEPREFIX  
    *NAMESPACEURI  
    *NODENAME  
    *NODEVALUE  
    *XPATH  
    *XPATH_CONCISE  
    xpath expression  
  
>--- NODESVALUE2 ----- *LOCALNAME -----  
----->  
    NODESVALUE3      *NAMESPACEPREFIX  
    NODESVALUE4      *NAMESPACEURI
```

NODESVALUE5 **NODENAME*
**NODEVALUE*
**XPATH*
**XPATH_CONCISE*
xpath expression

Keywords

NODES

The NODES keyword must specify an appropriately formulated XPath expression that identifies one or more nodes in the document for which values are to be returned.

The XPath expression is evaluated in the context of the "current" node. By default the "current" node is the document node but it may be altered. Refer to the SET command for information on setting the "current" node.

For more information about XPath expressions used with the XMLQueryService, refer to [Quick Guide to XPath expressions for use with XMLQueryService](#).

If no nodes are selected when the XPath expression is evaluated, it is not treated as an error. In this case, the command succeeds (with a returned status of 'OK') but the working list will be empty.

NODESVALUE1

The NODESVALUE1 keyword identifies the value that is to be returned in the FIRST column of the provided working list for each "selected" node (that is, each node "selected" by evaluating the XPath expression specified in the NODES keyword).

The keyword is optional. If the XPath expression specified in the NODES keyword fully identifies a set of elements or attributes whose value(s) are to be returned, there is no need to specify the NODESVALUE1 keyword. The default value of '.' is, in fact, an XPath expression that specifies that the value of the context node (in this case, each "selected" node) is to be used.

If you specify this keyword, you should specify a further XPath expression that will be evaluated in the context of each "selected" node and that will identify a single element

or attribute value relative to that node whose value is to be returned.

For more information about XPath expressions used with the XMLQueryService, refer to [Quick Guide to XPath expressions for use with XMLQueryService](#).

Alternatively, you may specify one of the following special values:

*LOCALNAME: returns the node name (usually an element or attribute name) of the "selected" node, without any namespace prefix, if present;

*NAMESPACEPREFIX: returns the namespace prefix of the "selected" node;

*NAMESPACEURI: returns the namespace URI of the "selected" node;

*NODENAME: returns the node name (usually an element or attribute name) of the "selected" node;

*NODEVALUE: returns the value of the "selected" node, equivalent to using an XPath expression of '.';

*XPATH:

*XPATH_CONCISE: These values both return a generated XPath expression that uniquely identifies the "selected" node within the XML document. The generated XPath expressions mostly use ordinal notation, and so are valid only for the specific node instance in the specific document instance. Such generated expressions can be used, however, to iteratively process a document using further QUERY commands. The second form generates an expression that is more concise (though less human-readable), particularly when the document is being processed in namespace-aware mode. The concise form may be necessary for use with very complex XML documents that may otherwise generate XPath expressions that are longer than can be processed by the client application.

NODESVALUE2 The keywords NODESVALUE2 through NODESVALUE5
NODESVALUE3 are optional, but, if specified, they function similarly to the

NODESVALUE4 NODESVALUE1 keyword.

NODESVALUE5 They allow you to specify further XPath expressions that (with the NODESVALUE1 keyword) identify up to five separate values relative to each "selected" node to be returned in the corresponding working list columns.

Unlike the NODESVALUE1 keyword, these keywords have no default. If you do not specify them, they will not be used and the corresponding working list columns are not referenced or required.

If you do specify these keywords, you must do so contiguously. The service will stop looking after the first keyword that is not used. For example, if you specify NODESVALUE1 and NODESVALUE3, then the latter will be ignored because NODESVALUE2 was not used.

Examples

The following examples execute a QUERY command that will query each <SalesOrder> element in the presently loaded document and, for each, return

- the value of the @SONumber attribute (sales order number) and
- the value of the @CustNumber attribute of the contained <Customer> element.

The two values are returned for each "selected" node in the entries of the ORDERS working list.

RDML Example:

```
DEFINE FIELD(#ORDERNUM) TYPE(*CHAR) LENGTH(10) COLHDG('C  
DEFINE FIELD(#CUSTNUM) TYPE(*CHAR) LENGTH(10) COLHDG('Cus  
DEF_LIST NAME(#ORDERS) FIELDS(#ORDERNUM #CUSTNUM) TYPE
```

...

```
CLR_LIST NAMED(#ORDERS)  
CHANGE FIELD(#JSMCMD) TO("query nodes(//SalesOrder) nodesvalue1(@  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
define field(#ORDERNUM) type(*CHAR) length(10) colhdg('Order' 'Number  
define field(#CUSTNUM) type(*CHAR) length(10) colhdg('Customer' 'Numb
```

```
def_list name(#ORDERS) fields(#ORDERNUM #CUSTNUM) type(*WORKI  
...
```

```
clr_list named(#ORDERS)  
#jsmcmdx := 'query nodes(//SalesOrder) nodesvalue1(@SONumber) nodesval  
use builtin(JSMX_COMMAND) with_args(#jsmcmdx) to_get(#jsmsts #jsmms
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSA Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM Message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

Required

```
SERVICE_GET ----- PROPERTY ----- value -----  
-|
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM Message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

Examples

The examples retrieve the value of the com.acme.property.message type (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html" will be returned to the application in this case):

```
com.acme.property.message=html
```

RDML Example:

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX Example:

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhdl #jsmcmd) to_get(#jsmsts #jsr
```

ILE RPG Example:

```
c          eval    jsmcmd = 'service_get'  
c          + ' property(com.acme.property.messageType)'  
c          callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

```
SERVICE_UNLOAD ----- no keywords -----  
-----|
```

Keywords

There are no SERVICE_UNLOAD keywords.

Examples

RDML Example:

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO
```

RDMLX Example:

```
use builtin(jsmx_command) with_args(#jsmhandle service_unload) to_get(#jsr
```

ILE RPG Example:

```
c          eval   jsmcmd = 'service_unload'  
c          callp  p_jsmcmd(jsmcmd:jsmsts:jmmmsg)
```

5.21.4 XMLQueryService Example

[RDML Example](#)

5.22 SOAPAgentService

Service Name: **SOAPAgentService**

The SOAPAgentService acts as a client to a remote Web Service. To use this service, you need to create a service class using the SOAP wizard.

The SOAPAgentService supports the following commands:

5.22.1 [SERVICE_LOAD](#)

5.22.2 [SERVICE_GET](#)

5.22.3 [OPEN](#)

5.22.4 [SET](#)

5.22.5 [CALL](#)

5.22.6 [GET](#)

5.22.7 [IS](#)

5.22.8 [CLOSE](#)

5.22.9 [SERVICE_UNLOAD](#)

5.22.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	TIMEZONE	value	Optional. See 5.1.3 Time Zones . This overrides the timezone service property. If no service property then the default TimeZone is used.

5.22.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.22.3 OPEN

The OPEN command loads the service class used to handle the SOAP transaction.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

OPEN	SERVICE	value	Mandatory. Service name.
------	---------	-------	--------------------------

The SERVICE keyword value is used in a resource property lookup using the property key service.'value' and service.archive.'value' to locate the binding class and the binding jar file that contains the binding class and supporting classes generated by the SOAPAgent Wizard. If the service binding archive entry does not exist then the binding jar file must exist in the JSM jar subdirectory instead of a user specified directory.

This example uses a service class named com.acme.SearchService for a SERVICE value of SEARCH:

```
service.search=com.acme.SearchService  
service.archive.search=bindings/search.jar
```

Example

```
OPEN SERVICE(SEARCH)
```

5.22.4 SET

The SET command sets the context and data of the operation parameters and fragments.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	OPERATION	value	Optional. Operation name.
	PARAMETER	value	Optional. Parameter name.
	FRAGMENT	value	Optional. Fragment name.
	LIST	value	Optional. List name.
	VALIDATE	*ALL	Optional. Validate SOAP request and response messages using an XML Schema. Used with the OPERATION keyword.
		*OPTIONAL	Only validate if service properties are available. Default.
		*NONE	No validation.
		*REQUEST	Validate SOAP request message.
		*RESPONSE	Validate SOAP response message.

Once the service has been opened, the program needs to specify which operation will be used, by calling the SET OPERATION command. The optional VALIDATE keyword controls whether XML Schema validation is done on the SOAP XML request and response messages. The external XML Schema files used to validate the SOAP messages is specified in the service properties file. The service and operation names are used in the service property to qualify

each entry.

Example

```
# service.validate.request.service.operation=soap-agent-schemas/soap-  
envelope.xsd  
# service.validate.response.service.operation=soap-agent-schemas/soap-  
envelope.xsd
```

Once the operation has been set, then the program needs to set the parameters and their values. If a parameter is simple and requires no fragments or lists, then a SET PARAMETER command will set the parameter value with the program field value.

If the parameter is a complex object, then the SET PARAMETER command needs to be called to set the context of future calls to act on this parameter. Several SET FRAGMENT and SET LIST commands might need to be called to fully populate the parameter object.

Call the SET PARAMETER command again to move onto creating the next parameter.

Once all parameters have been prepared, then execute the CALL command.

Example

```
SET OPERATION(KEYWORDSEARCHREQUEST)  
SET OPERATION(KEYWORDSEARCHREQUEST) VALIDATE(*RESPON  
SET PARAMETER(KEYWORDSEARCHREQUEST)  
SET LIST(AUTHORS) SERVICE_LIST(...)  
SET FRAGMENT(REQUEST) SERVICE_EXCHANGE(*FIELD)
```

Example SOAP/1.1 Envelope XMLSchema

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<!--
```

Schema for the SOAP/1.1 envelope

Portions © 2001 DevelopMentor, © 2001 W3C (Massachusetts Institute of Technology), © 2001 Institut National de Recherche en Informatique et en Automatique, Keio University

All Rights Reserved.

This document is governed by the W3C Software License [1] as described in

[1] <http://www.w3.org/Consortium/Legal/copyright-software-19980720>

[2] <http://www.w3.org/Consortium/Legal/IPR-FAQ-20000620.html#DTD>

By obtaining, using and/or copying this work, you (the licensee) agree that you have read and understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its documentation with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the software and documentation portions thereof, including modifications, that you make:

1. The full text of this NOTICE in a location viewable to users of the redistributed or derivative work.

2. Any pre-

existing intellectual property disclaimers, notices, or terms and conditions.

If none exist, a short notice of the following form (hypertext is preferred, text is also permitted) should be used within the body of any redistributed or derivative work. The notice shall read: "Copyright © 2001 World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. <http://www.w3.org/Consortium/Legal/>"

3. Notice of any changes or modifications to the W3C files, including the date and location of the changes. (We recommend you provide URIs to the location from which the code is distributed.)

Original W3C files; <http://www.w3.org/2001/06/soap-envelope>

Changes made:

- reverted namespace to <http://schemas.xmlsoap.org/soap/envelope/>
- reverted mustUnderstand to only allow 0 and 1 as lexical values
- made encodingStyle a global attribute 20020825
- removed default value from mustUnderstand attribute declaration

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND WITHOUT ANY REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING ANY WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR APPLICATION. NO DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, TRADE SECRETS, OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or promotional materials pertaining to the software without specific, written prior permission.

Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

-->

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
  targetNamespace="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:import1="http://soap.service.acme.com">

  <!-- Import SOAP Service XMLSchema EveryNumber -->
  <xs:import namespace="http://soap.service.acme.com" schemaLocation="eve

  <!-- Envelope -->
  <xs:element name="Envelope" type="tns:Envelope"/>
  <xs:complexType name="Envelope">
    <xs:sequence>
      <xs:element ref="tns:Header" minOccurs="0"/>
      <xs:element ref="tns:Body" minOccurs="1"/>
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" p
    </xs:sequence>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>

  <!-- Header -->
  <xs:element name="Header" type="tns:Header"/>
  <xs:complexType name="Header">
    <xs:sequence>
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" p
    </xs:sequence>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>
```

```

<!-- Body -->
<xs:element name="Body" type="tns:Body"/>
<xs:complexType name="Body">
  <xs:sequence>

<!--
  Accept any request, response or fault element
  <xs:any namespace="##any" minOccurs="0" maxOccurs="unbounded" pro
-->
  <xs:element ref="import1:getNumber" minOccurs="0"/>
  <xs:element ref="import1:getNumberResponse" minOccurs="0"/>
  <xs:element ref="tns:Fault" minOccurs="0"/>

  </xs:sequence>
  <xs:anyAttribute namespace="##any" processContents="lax"/>
</xs:complexType>

<!-- Global Attributes -->
<xs:attribute name="mustUnderstand">
  <xs:simpleType>
    <xs:restriction base="xs:boolean">
      <xs:pattern value="0|1"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute name="actor" type="xs:anyURI"/>

<xs:simpleType name="encodingStyle">
  <xs:list itemType="xs:anyURI"/>
</xs:simpleType>

<xs:attribute name="encodingStyle" type="tns:encodingStyle"/>
<xs:attributeGroup name="encodingStyle">
  <xs:attribute ref="tns:encodingStyle"/>
</xs:attributeGroup>

<!-- Fault -->
<xs:element name="Fault" type="tns:Fault"/>
<xs:complexType name="Fault" final="extension">

```

```

<xs:sequence>
  <xs:element name="faultcode" type="xs:QName"/>
  <xs:element name="faultstring" type="xs:string"/>
  <xs:element name="faultactor" type="xs:anyURI" minOccurs="0"/>
  <xs:element name="detail" type="tns:detail" minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<!-- Fault detail -->
<xs:complexType name="detail">
  <xs:sequence>
    <xs:any namespace="##any" minOccurs="0" maxOccurs="unbounded" pro
  </xs:sequence>
  <xs:anyAttribute namespace="##any" processContents="lax"/>
</xs:complexType>

</xs:schema>

```

Example SOAP Service XMLSchema

```

<?xml version="1.0" encoding="utf-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:tns="htt

  <xs:element name="getNumber">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="param1" type="tns:EveryNumber"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="getNumberResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="getNumberReturn" type="tns:EveryNumber"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

```

```
</xs:complexType>
</xs:element>

<xs:complexType name="EveryNumber">
  <xs:sequence>
    <xs:element name="n_double" type="xs:double"/>
    <xs:element name="n_float" type="xs:float"/>
    <xs:element name="n_int" type="xs:int"/>
    <xs:element name="n_long" type="xs:long"/>
  </xs:sequence>
</xs:complexType>

</xs:schema>
```

Example XMLSchema SimpleTypes

```
<xs:simpleType name="type-pattern">
  <xs:restriction base="xs:string">
    <xs:pattern value="[2-5][0-9]"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="type-enumeration">
  <xs:restriction base="xs:string">
    <xs:enumeration value="A"/>
    <xs:enumeration value="B"/>
    <xs:enumeration value="C"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="type-enumeration2">
  <xs:restriction base="xs:int">
    <xs:enumeration value="12"/>
    <xs:enumeration value="22"/>
    <xs:enumeration value="32"/>
  </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="one-hundred-or-more">
  <xs:restriction base="xs:int">
    <xs:minInclusive value="100"/>
  </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="one-hundred-or-less">
  <xs:restriction base="xs:int">
    <xs:maxInclusive value="100"/>
  </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="amount">
  <xs:restriction base="xs:decimal">
    <xs:totalDigits value="4"/>
    <xs:fractionDigits value="1"/>
    <xs:minInclusive value="22.0"/>
    <xs:maxInclusive value="80.5"/>
  </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="TimePeriod">
  <xs:restriction base="xs:string">
    <xs:pattern value="Days|Weeks|Months|Years"/>
  </xs:restriction>
</xs:simpleType>
```

5.22.5 CALL

The CALL command executes the operation.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CALL	USER	value	Optional. HTTP Basic Authentication.
	PASSWORD	value	Optional. HTTP Basic Authentication.
	URL	value	Optional. HTTP End point.
	FAULT	value	Optional. File path. If a SOAP Fault occurs then the SOAP message to written to the specified file. This file can then be processed by an XML service.
	VERSION		Optional. See VERSION .

If the response from the web service is a simple data type, then updating of the program with this value will happen automatically. If the response object is complex then several GET FRAGMENT and GET LIST commands will be required to receive all the returned data into the program.

Example

CALL

5.22.6 GET

The GET command retrieves the data from the response object returned by the CALL operation.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	FRAGMENT	value	Optional. Fragment name.
	LIST	value	Optional. List name.

Example

```
GET LIST(STATUS) SERVICE_LIST(...)  
GET FRAGMENT(RESPONSE) SERVICE_EXCHANGE(*FIELD)
```

5.22.7 IS

The IS command is used to check if the return parameter is a NULL reference. If the condition check is true then an OK status is returned or a status of NO is returned.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
IS	NULL	*RETURN	Conditional.
	NOT_NULL	*RETURN	Conditional.

Example

```
IS NULL(*RETURN)
IS NOT_NULL(*RETURN)
```

5.22.8 CLOSE

The CLOSE command releases resources and resets the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

CLOSE			
-------	--	--	--

Example

CLOSE			
-------	--	--	--

5.22.9 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.23 SOAPServerService

Service Name: **SOAPServerService**

The SOAPServerService receives a SOAP request and returns a SOAP response to the third-party client. To use this service, you need to create service and provider classes using the SOAP wizard.

The SOAPServerService supports the following commands:

5.23.1 [SERVICE_LOAD](#)

5.23.2 [SERVICE_GET](#)

5.23.3 [OPEN](#)

5.23.4 [IS](#)

5.23.5 [GET](#)

5.23.6 [SET](#)

5.23.7 [CLOSE](#)

5.23.8 [SERVICE_UNLOAD](#)

5.23.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	TIMEZONE	value	Optional. See 5.1.3 Time Zones . This overrides the timezone service property. If no service property then the default TimeZone is used.
	ARCHIVE		Optional. See ARCHIVE .
	ENCODING		Optional. See ENCODING .
		*BINARY	Archive content with no encoding changes.

If the ARCHIVE keyword is used the service will use the ENCODING keyword value if it is present or determine the encoding from the protocol content-type entry or use a default value to convert the received byte content to a Unicode string ready for the archiving stage.

The ENCODING keyword is only used when the service is going to archive the received content. The keyword value of *BINARY instructs the service to write the byte content unaltered to the archive file.

The ARCEncode keyword controls the archive file content encoding.

If the content type is **not** of type text or XML then the byte content is written to the file unaltered.

Using the keyword ENCODING (*BINARY) with the ARCHIVE keyword causes all received content to be archived unaltered.

5.23.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.23.3 OPEN

The OPEN command loads the service and provider classes to handle the SOAP transaction.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
OPEN	SERVICE	value	Mandatory. Service name.
	VALIDATE	*ALL	Optional. Validate SOAP request and response messages using an XML Schema.
		*OPTIONAL	Only validate if service properties are available. Default.
		*NONE	No validation.
		*REQUEST	Validate SOAP request message.
		*RESPONSE	Validate SOAP response message.

The SERVICE keyword is a symbolic name for the service and provider classes created by the SOAP Wizard. This example uses a service class named com.acme.SearchService and a provider class com.acme.SearchProvider for a SERVICE value of SEARCH:

```
service.search=com.acme.SearchService  
provider.search=com.acme.SearchProvider
```

The optional VALIDATE keyword controls whether XML Schema validation is done on the SOAP XML request and response messages. The external XML Schema files used to validate the SOAP messages is specified in the service properties file. The service and operation names are used in the service property to qualify each entry.

Example

```
# service.validate.request.service.operation=soap-server-schemas/soap-  
envelope.xsd  
# service.validate.response.service.operation=soap-server-schemas/soap-  
envelope.xsd
```

Example

OPEN SERVICE(SEARCH)

5.23.4 IS

The IS command is used to check if a parameter has a NULL reference. If the condition check is true then an OK status is returned or a status of NO is returned.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
IS	NULL	value	Conditional. Parameter Name.
	NOT_NULL	value	Conditional. Parameter Name.

Example

```
IS NULL(ID)  
IS NOT_NULL(SURNAME)
```

5.23.5 GET

The GET command retrieves the contents of the operation parameters.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	PARAMETER	value	Optional. Parameter name.
	FRAGMENT	value	Optional. Fragment name.
	LIST	value	Optional. List name.
	OBJECT	*OPERATION	Optional. Retrieve operation name.
	PROPERTY	value	Optional. HTTP protocol property.

Example

```
GET PARAMETER(ID)  
GET PROPERTY(SOAPACTION)
```

5.23.6 SET

The SET command is used to create the return parameter object.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	PARAMETER	*RETURN	Optional.
	FRAGMENT	value	Optional. Fragment name.
	LIST	value	Optional. List name.

If the return parameter is simple and requires no fragments or lists, then a SET PARAMETER(*RETURN) command will set the parameter value with the program field value.

If the parameter is a complex object, then the SET PARAMETER(*RETURN) command needs to be called to set the context of future calls to act on this parameter. Several SET FRAGMENT and SET LIST commands might need to be called to fully populate the return parameter object.

Example

```
SET PARAMETER(*RETURN)
```

5.23.7 CLOSE

The CLOSE command closes the open service and creates the SOAP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CLOSE	FAULT	value	Optional. Fault string.
	CONTENT		Optional. See CONTENT .
	ARCHIVE		Optional. See ARCHIVE .

Example

```
CLOSE  
CLOSE FAULT(Duplicate name)
```

5.23.8 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

DO NOT use the SERVICE_UNLOAD command when sending a HTTP/SOAP response. Use the CLOSE command to create the response and then use the JSM_CLOSE BIF to close the connection. The byte array response returned to JSM Client program by the CLOSE command will be become the HTTP/SOAP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.24 XMLQueueService

Service Name: **XMLQueueService**

The XMLQueueService service allows XML messages to be received from and sent to MQSeries message queues and IBM i Data queues.

The following service property entries can be used to control the data encoding used when data is received from the queue and sent to the queue. When data is put to an IBM i data queue the data is padded with EBCDIC spaces. It is recommended that dq.put.encoding and dq.get.encoding values belong to the EBCDIC family. For example Cp037, Cp273 and Cp500 are valid values. By default the CCSID of the client program or the message is used to determine the encoding value.

dq.get.encoding

dq.put.encoding

mq.get.encoding

mq.put.encoding

The following service property entries can be used to stop the XML declaration being modified to suit the data encoding. The default value for these properties is *yes.

dq.put.modify.declaration=*no

mq.put.modify.declaration=*no

The following service property entries can be used to explicitly specify the XML declaration IANA encoding used by the XML declaration modification process.

If these service properties are missing then the IANA encoding is determined from the encoding.

dq.put.encoding.declaration= Any valid IANA encoding. ie iso-8859-1

mq.put.encoding.declaration= Any valid IANA encoding. ie iso-8859-1

IBM Java Unicode to EBCDIC byte encoders use the EBCDIC NL (New Line 0x15) character instead of the EBCDIC LF (Line Feed 0x25) character. When the Unicode XML result is converted to the selected encoding and this encoding is EBCDIC, the New Line character will be replaced with a Line Feed character. The default value for these properties is *no.

dq.put.ebcdic.nl2lf=*yes

mq.put.ebcdic.nl2lf=*yes

The XMLQueueService can be configured to handle text messages. If the keyword MESSAGE(*TEXT) is present on the SERVICE_LOAD command then the service is expecting text messages and the message value is wrapped with <text> and </text> tags.

```
<?xml version="1.0" encoding="utf-8"?>  
<text>The quick brown fox jumped over the lazy dogs</text>
```

The XMLQueueService supports the following commands:

[5.24.1 SERVICE_LOAD](#)

[5.24.2 SERVICE_GET](#)

[5.24.3 RECEIVE](#)

[5.24.4 SEND](#)

[5.24.5 ARCHIVE](#)

[5.24.6 SERVICE_UNLOAD](#)

Refer to [5.1.13 MQSeries Built-In Functions](#) for details of the MQSeries Built-In Functions.

5.24.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	SCHEMA		Optional. See SCHEMA .
	VALIDATING		Optional. See VALIDATING .
	DOMSET		Optional. See DOMSET .
	DOMSETMODEL		Optional. See DOMSETMODEL .
	DOMSETRESULT		Optional. See DOMSETRESULT .
	DOMGET		Optional. See DOMGET .
	DOMGETRESULT		Optional. See DOMGETRESULT .
	MESSAGE		*TEXT Optional. Service is expecting text messages.

5.24.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.24.3 RECEIVE

When the RECEIVE command of this service is executed the following steps occur:

If there is no content, then an OK status is returned.

The service then transforms the XML and binds the data into the program fields and list.

The receive command can be called more than once to receive data from a complex XML document.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

RECEIVE	XSL		Mandatory. See XSL .
---------	-----	--	--------------------------------------

5.24.4 SEND

When the SEND command of this service is executed the following steps occur:

The service using the XSL keyword transforms the program fields and list into the resultant XML. If a FRAGMENT keyword is present this transformation is stored, as this is only a fragment of the final XML to be sent. When no FRAGMENT keyword is present this transformation is treated as the last and the final output XML is constructed and ready to put on a queue.

The XMLQueueService can send text messages, if the result XML uses the text tags to wrap the text message and the MESSAGE(*TEXT) keyword is present on the final SEND command, these tags are removed and the text value is sent.

```
<?xml version="1.0" encoding="utf-8"?>
<text>The quick brown fox jumped over the lazy dogs</text>
```

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	XSL		Mandatory. See XSL
		*COPY	Copies the input source to the output source.
	FRAGMENT		Optional. See FRAGMENT .
	OUTPUT		Optional. See OUTPUT .
	MESSAGE	*TEXT	Optional. Send text message.

5.24.5 ARCHIVE

The ARCHIVE command saves the selected source to the archive file using the specified encoding.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
ARCHIVE	FILE	value	Mandatory. File path.
	ENCODING		Optional. See ENCODING .
	SOURCE	*RECEIVE	Optional. Archive receive source. Default. *RECEIVE.
		*SEND	Archive send source.

5.24.6 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.25 XMLBindQueueService

Service Name: **XMLBindQueueService**

The XMLBindQueueService service allows XML messages to be received from and sent to MQSeries message queues and IBM i Data queues.

The following service property entries can be used to control the data encoding used when data is received from the queue and send to the queue. When data is put to an IBM i data queue the data is padded with EBCDIC spaces. It is recommended that dq.put.encoding and dq.get.encoding values belong to the EBCDIC family. For example Cp037, Cp273 and Cp500 are valid values. By default the CCSID of the client program or message is used to determine the encoding value.

dq.get.encoding

dq.put.encoding

mq.get.encoding

mq.put.encoding

The following service property entries can be used to explicitly specify the XML declaration IANA encoding used by the XML declaration modification process.

If these service properties are missing then the IANA encoding is determined from the encoding.

dq.put.encoding.declaration= Any valid IANA encoding. ie iso-8859-1

mq.put.encoding.declaration= Any valid IANA encoding. ie iso-8859-1

The following service property entries can be used to stop the XML declaration being modified to suit the data encoding. The default value for these properties is *yes.

dq.put.modify.declaration=*no

mq.put.modify.declaration=*no

IBM Java Unicode to EBCDIC byte encoders use the EBCDIC NL (New Line 0x15) character instead of the EBCDIC LF (Line Feed 0x25) character. When the Unicode XML result is converted to the selected encoding and this encoding is EBCDIC, the New Line character will be replaced with a Line Feed character. The default value for these properties is *no.

dq.put.ebcdic.nl2lf=*yes

mq.put.ebcdic.nl2lf=*yes

The XMLBindQueueService supports the following commands:

5.25.1 [SERVICE_LOAD](#)

5.25.2 [SERVICE_GET](#)

5.25.3 [BIND](#)

5.25.4 [CLOSE](#)

5.25.5 [GET](#)

5.25.6 [SET](#)

5.25.7 [WRITE](#)

5.25.8 [ARCHIVE](#)

5.25.9 [SERVICE_UNLOAD](#)

Refer to [5.1.13 MQSeries Built-In Functions](#) for details of the MQSeries Built-In Functions.

5.25.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	SCHEMA		Optional. See SCHEMA .
	VALIDATING		Optional. See VALIDATING .
	DOMSET		Optional. See DOMSET .

5.25.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.25.3 BIND

The BIND command binds the specified service code to the loaded inbound document or creates a new empty outbound document.

The SERVICE keyword value is used in a resource property lookup using the property key service.'value' and service.archive.'value' to locate the binding class and the binding jar file that contains the binding class and supporting classes generated by the XMLBinding Wizard. If the service binding archive entry does not exist then the binding jar file must exist in the JSM jar subdirectory instead of a user specified directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
BIND	SERVICE	value	Service name.
	TYPE	*INBOUND	Conditional. Inbound binding.
		*OUTBOUND	Conditional. Outbound binding.
	BINDTRACE	*YES	Optional. Trace the inbound bind result.
		*NO	Default.
FILTER		Optional. See FILTER .	

5.25.4 CLOSE

The CLOSE command closes the current BIND. The binding can be closed before or after the MQ put or DQ put commands. If the binding is closed after the put, then the output content is released for garbage collection.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

CLOSE			
-------	--	--	--

5.25.5 GET

The GET command is used to read the inbound document object.

Fragments use only fields, so the GET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the GET LIST command requires a working list argument.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	INSTRUCTION	value	Optional. Get XML processing instruction.

5.25.6 SET

The SET command is used to populate the outbound document object.

Fragments use only fields, so the SET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the SET LIST command requires a working list argument.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	INSTRUCTION	value	Optional. Set XML processing instruction.
	DATA	value	Conditional. PI data.

5.25.7 WRITE

The WRITE command serializes the document object in preparation for the MQ put or DQ put commands.

The MQ put and or DQ put can be called multiple times sending the same content to different queues.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
WRITE	INDENT	*YES	Optional. Indent XML content.
		*NO	Default.
	INDENT-AMOUNT	value	Optional. Indent XML amount. Default. 0. If the amount is not zero then indent is enabled.
	DOCTYPE	value	Optional. Include XML <!DOCTYPE> element.
	PUBLIC	value	Conditional. The optional public component of the XML DOCTYPE declaration.
	OMIT-DECLARATION	*YES	Optional. Omit the XML declaration.
		*NO	Default.
	BINDTRACE	*YES	Optional. Trace the outbound bind result.
		*NO	Default.
	FILTER		Optional. See FILTER .

5.25.8 ARCHIVE

The ARCHIVE command saves the selected source to the archive file using the specified encoding.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
ARCHIVE	FILE	value	Mandatory. File path.
	ENCODING		Optional. See ENCODING .
	SOURCE	*RECEIVE	Optional. Archive receive source. Default. *RECEIVE.
		*SEND	Archive send source.

5.25.9 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.26 FileQueueService

Service Name: **FileQueueService**

The FileQueueService service allows files be received from and sent to MQSeries message queues and IBM i Data queues.

The following service property entries can be used to control the data encoding used when data is received from the queue and send to the queue. When data is put to an IBM i data queue the data is padded with EBCDIC spaces. It is recommended that dq.put.encoding and dq.get.encoding values belong to the EBCDIC family. For example Cp037, Cp273 and Cp500 are valid values. By default the CCSID of the client program or the message is used to determine the encoding value.

dq.get.encoding

dq.put.encoding

mq.get.encoding

mq.put.encoding

The following service property entries can be used to stop the XML declaration being modified to suit the data encoding. The default value for these properties is *yes.

dq.get.modify.declaration=*no

dq.put.modify.declaration=*no

mq.get.modify.declaration=*no

mq.put.modify.declaration=*no

The following service property entries can be used to explicitly specify the XML declaration IANA encoding used by the XML declaration modification process.

If these service properties are missing then the IANA encoding is determined from the encoding.

dq.get.encoding.declaration= Any valid IANA encoding. ie iso-8859-1

dq.put.encoding.declaration= Any valid IANA encoding. ie iso-8859-1

mq.get.encoding.declaration= Any valid IANA encoding. ie iso-8859-1

mq.put.encoding.declaration= Any valid IANA encoding. ie iso-8859-1

IBM Java Unicode to EBCDIC byte encoders use the EBCDIC NL (New Line 0x15) character instead of the EBCDIC LF (Line Feed 0x25) character. When

the Unicode XML result is converted to the selected encoding and this encoding is EBCDIC, the New Line character will be replaced with a Line Feed character. The default value for these properties is *no.

dq.put.ebcdic.nl2lf=*yes

mq.put.ebcdic.nl2lf=*yes

The FileQueueService supports the following commands:

[5.24.1 SERVICE_LOAD](#)

[5.24.2 SERVICE_GET](#)

[5.26.3 READ](#)

[5.26.4 WRITE](#)

[5.24.6 SERVICE_UNLOAD](#)

Refer to [5.1.13 MQSeries Built-In Functions](#) for details of the MQSeries Built-In Functions.

5.26.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD			
--------------	--	--	--

5.26.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.26.3 READ

The READ command reads the specified file and internally stores the content in Unicode form.

The internal Unicode content is sent to a Data Queue or Message Queue using the JSMDQPUT and JSMQPUT programs.

Refer to [5.1.13 MQSeries Built-In Functions](#) for details of the MQSeries Built-In Functions.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
READ	FILE	value	Mandatory. File path.
	ENCODING	value	Optional. Default is UTF8.

5.26.4 WRITE

The WRITE command saves the internally stored Unicode content to the specified file.

The internal Unicode content is received from a Data Queue or Message Queue using the JSMDQGET and JSMMQGET programs.

Refer to [5.1.13 MQSeries Built-In Functions](#) for details of the MQSeries Built-In Functions.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
WRITE	FILE	value	Mandatory. File path.
	ENCODING	value	Optional. Default is UTF8.

5.26.5 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.27 OpenLDAPService

The OpenLDAPService allows you to develop applications that can access LDAP compliant directories.

LDAP (Lightweight Directory Access Protocol) is a protocol that is becoming more and more prevalent in databases and systems throughout the world. It is a protocol that is widely accepted as the industry standard for deploying directory based applications and solutions.

LDAP is typically used for look up directories (or databases) and is ideal for use in the following hyperthetical examples:

- You need to regularly access relatively static data such as telephone and email directories. For example, email addresses and phone numbers do not change very often, but might be accessed thousands of times a day by your users.
- You need to access data that is primarily read only. LDAP compliant directories are optimized more for the fast read of information than for the updating, adding or deleting of data. Once again, your users might regularly access email addresses from your central system, but this information will rarely be updated. Alternatively, you might have a directory containing information on all the printers connected to your network.
- You need to deploy distributed databases, where the reference information might be kept on multiple servers. Such architecture can be useful if, for example, one server is down, then the necessary information could be picked up off another server.
- You want to set up a system to bypass the multiple Logons that users have to make to access information from different systems. By applying a single Logon across the board, you could significantly reduce the workload placed on your help desk by users forgetting their passwords.

A classic example of where an LDAP directory might be used is in large organizations with a large employee directory. This directory might contain staff contact information that is accessible by staff and perhaps by partners and customers. This directory of contact information is not changed very often, so is used primarily for heavy read access by numerous systems (for example, your email system).

Unique objects within an LDAP directory tree are referenced by what is known as their distinguished name (DN). An object within a tree has a relative distinguished name (RDN), which identifies the object relative to its parents. A

DN may therefore be defined as a collection of RDNs that, when combined, uniquely identify the DN within the entire directory. RDN's are generally of the form attribute=value, where attributes might be c (country), cn (common name), o (organization) and so forth. The DN is constructed by combining the RDNs of the object with those of its parents. For example, the DN of the root of the sample tree might be ou=Sales Team, o=LANSA, c=au (this uniquely defines the Sales Team and the company, LANSA, in Australia). The RDN of a member of the sales team is cn=Terry Briggs (the common name of the salesperson concerned is Terry Briggs). Therefore the DN, or the distinguished name, of the directory object that uniquely defines that Terry Briggs as a member of the sales team in LANSA Australia is cn=Terry Briggs, ou=Sales Team, o=LANSA, c=au.

LANSA's OpenLDAPService is designed to enable you to interface easily with LDAP directories from within your application. While the main purpose of LDAP directories is for read operations, there will be times when you will want to update, create or delete entries. The OpenLDAPService provides you with the facilities to do these, should you need to.

By using the OpenLDAPService, LANSA will handle all the interface requirements to the LDAP Server, saving you having to learn how to do so yourself. The service provides the following commands to enable you to perform the standard transactions that can be performed on an LDAP directory:

Get	Perform a single record fetch on a DN in the directory
Add	Add a single record DN to the directory
Modify	Attribute values of a specific DN
Delete	Delete a DN record from the directory
Search	Search through multiple DN records in the directory.

The OpenLDAPServer also provides the necessary commands to enable your application to BIND to and UNBIND from the server. The term BIND is an industry standard term that refers to the way applications connect to an LDAP server.

Prior to using this service you will need to understand how to set up and use an LDAP server on the IBM i. More information can be found on this topic in the IBM i manuals.

Related Services

The OpenLDAPService is not dependant on any other services. Using this service alone, you can read, add, modify and delete entries in your LDAP directory.

Technical Specifications

By default, the OpenLDAPService uses port 389. For secure connections it uses port 636.

5.27.1 What can I use the OpenLDAPService for?

LDAP servers are used to make commonly used directory information more easily available across multiple applications that could be running on multiple servers. The OpenLDAPService is designed to help you interface with LDAP servers more easily.

Following are two examples of where you might use the OpenLDAPService.

To access an LDAP directory containing company contact details

A large organization may have multiple applications that each contains their own repository of employee contact information, often in duplication. To overcome this, they may decide to create an LDAP directory of contact information to centralize this data. Every application that needs to access employee contact information could access this LDAP directory. By using the OpenLDAPService you could very easily write the code required to access this repository of contact information for any number of your applications. These client applications could be running on different servers.

To access an LDAP directory containing User ID, Password, and Access Rights

We are all familiar with holding different User Ids and passwords for the different applications that we use. And we are also familiar with the tedious process of resetting those passwords because we have forgotten what they are. To reduce this problem, an organization may decide to create a central repository of users and their passwords and access rights in an LDAP directory so this information is easier to maintain and to control. You could then use the OpenLDAPService to authenticate your user Logons against this LDAP directory instead of doing so against an isolated local repository.

5.27.2 Using the OpenLDAPService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, Cobol, or C, you have to complete the same basic steps.

For example, an application that is going to fetch the employee phone number from an LDAP server would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

BIND

GET

UNBIND

SERVICE_UNLOAD

JSM(X)_CLOSE

The steps to add, change and delete entries in the directory would be very similar, but using the ADD, MODIFY or DELETE commands in place of the GET command.

Refer to the [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.27.3 OpenLDAPService Commands

Your application issues commands to the OpenLDAPService by passing the command strings through the Java Service Manager using the JSM_COMMAND or the JSMX_COMMAND Built-In Function, or an API for your chosen development language.

The commands that the OpenLDAPService processes are:

SERVICE_LOAD

BIND

GET

ADD

MODIFY

DELETE

SEARCH

UNBIND

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
---->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

SERVICE The name of the service to be loaded - in this case OpenLDAPService.

TRACE To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.

The possible values for the TRACE keyword are:

- *NO
- *YES
- *ERROR

The trace option *ERROR will turn on tracing and if the

service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.

TRACE_NAME This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(OpenLDAP;  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX;

```
#jxmcommand := 'service_load service(OpenLDAPService)'  
use builtin(jsmx_command) with_args(#jmxh1 #jxmcommand) to_get(#jsn
```

BIND

The BIND command is responsible for establishing a connection to the LDAP server.

A simple bind sends the user's DN and password. An anonymous bind may be established by not sending a password.

It is possible to establish a secure connection, using SSL, and this BIND command provides for this option.

```

                                Required
BIND ----- HOST ----- host ----->

                                Optional

>-- PORT ----- port ----->

>-- SECURE ----- *NO ----->
                        *YES

>-- DN ----- authentication name ----->

                                Conditional

>-- PASSWORD ----- value -----|
```

Keywords

- | | |
|--------|--|
| HOST | The host address of the LDAP server you connecting to.
This keyword is mandatory. |
| PORT | The port to connect to on the host.
By default, this value will be 389 for a simple connection, or 636 for a secured connection.
This keyword is optional. |
| SECURE | Used to specify whether you want to establish a secure connection to the server. |

The default value is *NO (for an unsecured connection).
This keyword is optional.

DN The authentication name.
The DN may be used without a password, establishing an anonymous connection. It is also possible not to pass this value at all, also establishing an anonymous connection.
This keyword is optional.

PASSWORD This keyword must be passed, along with the DN keyword, if authentication is required.
A DN value may be sent without a PASSWORD value, but a PASSWORD value may not be sent without a DN value.

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO("BIND HOST(<LDAP-server-  
name>) DN(cn=Administrator) PASSWORD(password)")  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#JSMCMD := 'Bind Host(<LDAP-server-  
name>) DN(cn=Administrator) Password(password)'  
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

GET

This command is used to GET an LDAP entry.

```
GET ----- DN ----- entry name ----->
                                     Optional
>-- ATTRIBUTES ----- *ALL -----|
                                     attribute names
```

Keywords

- DN** Specifies the distinguished name (DN) of the entry you are looking for.
This keyword is mandatory.
- ATTRIBUTES** This keyword specifies the attributes that you want returned. You may specify either *ALL (which will return all the attributes), or a comma delimited list of the attribute names. The default value is *ALL.
This keyword is optional.

Lists and Variables

The application must supply a working list into which the GET command will return the retrieved attributes. The working list must contain either one or two fields as follows:

- The first field is mandatory and will contain the attribute name for each retrieved attribute.
Suggested field length: you will need to ensure that the field is long enough to hold the longest possible attribute name for your LDAP Server. The field length will be character.
- The second field is optional. If supplied, it will contain the attribute value for each retrieved attribute.
Suggested field length: you will need to ensure that the field is long enough to hold the longest possible attribute value for your LDAP Server. The field length will be character.

Refer to the following examples to see how this works. For information on how

to supply a working list to service commands from RDML, RDMLX or 3GL applications, refer to [Java Service Manager Clients](#).

Examples

RDML

```
DEFINE FIELD(#ATNAME) TYPE(*CHAR) LENGTH(050)
DEFINE FIELD(#ATVALUE) TYPE(*CHAR) LENGTH(050)
DEF_LIST NAME(#WRKLIST) FIELDS(#ATNAME #ATVALUE) TYPE(*W
CHANGE FIELD(#JSMCMD) TO("GET DN(cn=John, cn=users, o=ibmteldir
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
Define Field(#ATNAME) Type(*CHAR) Length(050)
Define Field(#ATVALUE) Type(*CHAR) Length(050)
Def_List Name(#WRKLIST) Fields(#ATNAME #ATVALUE) Type(*Working)
#JSMCMD := 'Get DN(cn=John, cn=users, o=ibmteldir) Service_List(ATNAM
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

ADD

This command allows you to add an LDAP entry.

```
ADD ----- DN ----- entry name -----|
```

Keywords

DN The entry name you want to add.

This keyword is mandatory.

Lists and Variables

The application must supply a working list from which the ADD command will pass the new attributes. The working list must contain either one or two fields as follows:

- The first field is mandatory and will contain the attribute name for each added attribute.
Suggested field length: you will need to ensure that the field is long enough to hold the longest possible attribute name for your LDAP Server. The field length will be character.
- The second field is optional. If supplied, it will contain the attribute value for each added attribute.
Suggested field length: you will need to ensure that the field is long enough to hold the longest possible attribute value for your LDAP Server. The field length will be character.

Refer to the following examples to see how this works. For information on how to supply a working list to service commands from RDML, RDMLX or 3GL applications, refer to [Java Service Manager Clients](#).

Examples

RDML

```
DEFINE FIELD(#ATNAME) TYPE(*CHAR) LENGTH(050)  
DEFINE FIELD(#ATVALUE) TYPE(*CHAR) LENGTH(050)  
DEF_LIST NAME(#WRKLST) FIELDS(#ATNAME #ATVALUE) TYPE(*W  
CHANGE FIELD(#JSMCMD) TO("ADD DN(cn=John, cn=users, o=ibmteldi  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
Define Field(#ATNAME) Type(*CHAR) Length(050)
Define Field(#ATVALUE) Type(*CHAR) Length(050)
Def_List Name(#WRKLST) Fields(#ATNAME #ATVALUE) Type(*Working)
#JSMCMD := 'Add DN(cn=John, cn=users, o=ibmteldir) Service_List(ATNAME)
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

MODIFY

You will use this command to modify an LDAP entry. This command has three operations that may be performed - add, delete, and replace. It is important to note that these operations occur at the attribute level and not the DN record level. If you need to add or delete an entire DN then you should use the ADD or DELETE commands.

```
MODIFY ----- DN ----- entry name -----|
```

Keywords

DN The entry name that is to be modified.

This keyword is mandatory.

Lists and Variables

The application must supply a working list from which the MODIFY command will pass the modified attributes. The working list must contain three fields as follows:

- The first field is mandatory and will contain the operation to be performed.
Suggested field length: Character field of length 1.
The three possible values for this field are:
 - A for Add
 - D for delete
 - R for Replace.
- The second field is mandatory and will contain the attribute name for each modified attribute.
Suggested field length: you will need to ensure that the field is long enough to hold the longest possible attribute name for your LDAP Server. The field length will be character.
- The third field is mandatory. It will contain the attribute value for each modified attribute.
Suggested field length: you will need to ensure that the field is long enough to hold the longest possible attribute value for your LDAP Server. The field length will be character.

Refer to the following examples to see how this works. For information on how

to supply a working list to service commands from RDML, RDMLX or 3GL applications, refer to [Java Service Manager Clients](#).

Examples

RDML

```
DEFINE FIELD(#ATMOD) TYPE(*CHAR) LENGTH(001)
DEFINE FIELD(#ATNAME) TYPE(*CHAR) LENGTH(050)
DEFINE FIELD(#ATVALUE) TYPE(*CHAR) LENGTH(050)
DEF_LIST NAME(#WRKLST) FIELDS(#ATMOD #ATNAME #ATVALUE)
CHANGE FIELD(#JSMCMD) TO("MODIFY DN(cn=John, cn=users, o=ibm
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
Define Field(#ATMOD) Type(*CHAR) Length(001)
Define Field(#ATNAME) Type(*CHAR) Length(050)
Define Field(#ATVALUE) Type(*CHAR) Length(050)
Def_List Name(#WRKLST) Fields(#ATMOD #ATNAME #ATVALUE) Type(
#JSMCMD := 'Modify DN(cn=John, cn=users, o=ibmteldir) Service_List(ATM
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

DELETE

You will use this command to delete a DN record.

```
DELETE ----- DN ----- entry name -----|
```

Keywords

DN The entry name of the DN you wish to delete.

This keyword is mandatory.

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO("DELETE DN(cn=John, cn=users, o=ibmt  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#JSMCMD := Delete DN(cn=John, cn=users, o=ibmteldir) '
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

SEARCH

You will use this command to search for entries on the LDAP server.

```

                                Required
SEARCH ----- DN ----- entry name ----->

>-- SCOPE ----- *SUB ----->
      *ONE
      *BASE

                                Optional

>-- FILTER ----- *NONE -----|
      value
```

Keywords

- DN** This keyword will be used to define the base DN to search for.
This keyword is mandatory.
- SCOPE** This keyword defines the scope, or range, of the entries to search.
There are three possible values:
***SUB** - run the search over the base DN and all its entries within its sub-trees.
***ONE** - run the search over the entries under the base DN.
***BASE** - run the search over the base DN alone.
***SUB** is the default value.
This keyword is optional.
- FILTER** This keyword can be used to specify a filter for this search criteria.
The default value is ***NONE**.
This keyword is optional.

Lists and Variables

The application must supply a working list to which the SEARCH command

will pass the retrieved DN values. The working list must contain one field as follows:

- The field is mandatory and will contain the DN value for each retrieved DN. Suggested field length: you will need to ensure that the field is long enough to hold the DN value of your LDAP Server. The field will be character.

Refer to the following examples to see how this works. For information on how to supply a working list to service commands from RDML, RDMLX or 3GL applications, refer to [Java Service Manager Clients](#).

Examples

RDML

```
DEFINE FIELD(#DN) TYPE(*CHAR) LENGTH(050)
DEF_LIST NAME(#WRKLST) FIELDS(#DN) TYPE(*WORKING)
CHANGE FIELD(#JSMCMD) TO("SEARCH DN(o=ibmteldir) FILTER(obje
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
Define Field(#DN) Type(*CHAR) Length(050)
Def_List Name(#WRKLST) Fields(#DN) Type(*Working)
#JSMCMD := Search DN(o=ibmteldir) Filter(objectclass=person) Scope(*SUI
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

UNBIND

This command is used to disconnect the client from the LDAP server

```
UNBIND ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO(UNBIND)  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#JSMCMD := UnBind  
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service and removes the temporary directory.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 service_unload) to_get(#jsi
```

5.27.4 OpenLDAPService Examples

Examples are supplied for the most common client programming languages:

[RDML](#)

[RDMLX](#)

5.28 JMSFileService

Service Name: **JMSFileService**

The JMSFileService allows files to be sent and received using enterprise messaging systems such as ActiveMQ, SonicMQ, TibcoMQ and WebSphereMQ.

This service uses the Java Message Service JMS 1.1 specification.

Refer to <http://www.oracle.com/technetwork/java/jms/index.html>.

If you're using:

- ActiveMQ, then you require the ActiveMQ JMS jar files.
- SonicMQ, then you require the SonicMQ JMS jar files.
- TibcoMQ then you require the ActiveMQ JMS jar files.
- WebSphereMQ, then you require the IBM WebSphereMQ JMS jar files.

You will need to know how to use the enterprise messaging system selected for use with this JMSFileService. The setting up and configuring of either messaging system is beyond the scope of this guide.

The JMSFileService supports the following commands:

[5.28.1 SERVICE_LOAD](#)

[5.28.2 SERVICE_GET](#)

[5.28.3 CONNECT](#)

[5.28.4 SEND](#)

[5.28.5 RECEIVE](#)

[5.28.6 COMMIT](#)

[5.28.7 ROLLBACK](#)

[5.28.8 CLOSE](#)

[5.28.9 SERVICE_UNLOAD](#)

5.28.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD			
--------------	--	--	--

5.28.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.28.3 CONNECT

The CONNECT command is used to open a connection to the enterprise messaging system.

The currently available vendor implementations are ActiveMQ, SonicMQ, TibcoMQ and WebSphereMQ.

It is possible to create more vendor implementations by writing a Java class that implements the JMSVendor interface.

The JMSFileService does a service properties lookup using the VENDOR keyword value to determine which Java class will be used to create the connection object.

```
vendor.activemq=com.lansa.service.JMSVendorActiveMQ
vendor.sonicmq=com.lansa.service.JMSVendorSonicMQ
vendor.tibcomq=com.lansa.service.JMSVendorTibcoMQ
vendor.webspheremq=com.lansa.service.JMSVendorMQSeries
```

Keyword values can also be passed to the CONNECT command by using a working list argument. The first column supplies the keyword name and the second column the keyword value.

The command keyword takes precedence over the working list keyword value entry.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword
CONNECT	VENDOR	Value	Mandatory. Message System Vendor.
	HOST	value	Mandatory. hostname hostname:port.
	PORT	value	Optional. Default. 61616 for ActiveMQ. Default. 2506 for Soni Default. 7222 for Tibc Default. 1414 for

		WebSphereMQ.
USER	value	Optional.
PASSWORD	value	Optional.
CLIENTID	value	Optional.
QUEUE	value	Mandatory.
REPLYTO	*TEMPORARY	Optional. Create and use temporary reply queue
CHANNEL	value	Conditional. Required WebSphereMQ connection
QUEUE-MANAGER	value	Optional. Used by WebSphereMQ connection
CIPHER-SUITE	value	Optional. Used by WebSphereMQ connection
MESSAGE-PRIORITY	value	0 (lowest) to 9 (highest). Default. 4.
MESSAGE-DELIVERY	*PERSISTENT *NONPERSISTENT	Default. *PERSISTENT
MESSAGE-TIME-TO-LIVE	value	Default. 0 milliseconds. A value of 0 means keep forever.
SESSION-TRANSACTION	*YES *NO	Default. *NO.
SESSION-ACKNOWLEDGE	*AUTO *CLIENT *DUPOK	Default. *AUTO.

Example

```
CONNECT VENDOR(WEBSPHEREMQ) HOST(LOCALHOST) CHANNEL  
MANAGER(USERAGENT.QUEUE.MANAGER) QUEUE(USERAGENT.QI
```

```
CONNECT VENDOR(ACTIVEMQ) HOST(LOCALHOST) QUEUE(QUEUI
```

5.28.4 SEND

The SEND command is used to optionally send a file as a message. If no file is specified then an empty message is sent.

To include JMS message properties with the message use the optional working list argument.

The first column is the property type, the second column is the property name and the third column is the property value. Property types are I for integer, B for Boolean, S for string, H for short, L for long, D for decimal, F for float and J for special JMS properties.

The following type J properties use individual method calls on the JMS message object instead of the generic message.setStringProperty, message.setIntProperty and message.setBooleanProperty method calls.

- JMSTYPE (message.setJMSType)
- JMSMESSAGEID (message.setJMSMessageID)
- JMSCORRELATIONID (message.JMSCorrelationID)
- JMSPRIORITY (message.setJMSPriority)
- JMSREDELIVERED (message.setJMSRedelivered)
- JMSEXPIRATION (message.setJSMExpiration)
- JMSTIMESTAMP (message.setJMSTimestamp)
- JMSDELIVERYMODE (message.setJMSDeliveryMode)
- JMSREPLYTO (message.setJMSReplyTo)
- JMSDESTINATION (message.setJMSDestination)

If the message type is *TEXT then the encoding value is used to convert the file content into a Unicode string and the message will be sent as a JMS TextMessage object.

The message identifier (MessageID) is returned in the JSMMSG field.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	FILE	value	Optional. File path.

MESSAGE	*BINARY	Message type. JMS BytesMessage.
	*TEXT	Default. JMS TextMessage.
ENCODING	value.	Conditional. Used if message type is *TEXT Default. UTF-8.
ARCHIVE	value	Optional. Archive file.

Example

```
SEND MESSAGE(*TEXT) ENCODING(UTF-  
8) FILE(order.xml) SERVICE_LIST(PRPTYP,PRPNME,PRPVAL)
```

```
SEND MESSAGE(*BINARY) FILE(order.xls) SERVICE_LIST(PRPTYP,PRI
```

5.28.5 RECEIVE

The RECEIVE command is used to receive a message and optionally save the contents to a file.

To receive JMS message properties from the message use the optional working list argument.

The first column receives the property type, the second column receives the property name and the third column receives the property value. The property type is always S (String) for user-defined properties and J for special JMS properties.

If the received message is a TextMessage then the encoding value is used to covert the Unicode content into a byte content to be saved to the specified file.

The message identifier (MessageID) is returned in the JSMMSG field.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	FILE	value	Optional. File path.
	ENCODING	value	Conditional. Used if received message is a Unicode TextMessage. Default. UTF-8.
	WAITTIME	value.	Optional. Wait time in milliseconds. Default. 0 forever.
	SELECTOR	value	Optional. Message Selector.
	ARCHIVE	value	Optional. Archive file.

Example

```
RECEIVE ENCODING(UTF-8) FILE(order.xml) SERVICE_LIST(PRPTYP,PRPNME,PRPVAL)
```

5.28.6 COMMIT

The COMMIT command is used to commit the current transacted session. Use the CONNECT command SESSION-TRANSACTIONED keyword to start a transacted session.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

COMMIT			
--------	--	--	--

5.28.7 ROLLBACK

The ROLLBACK command is used to rollback the current transacted session. Use the CONNECT command SESSION-TRANSACTIONED keyword to start a transacted session.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

ROLLBACK			
----------	--	--	--

5.28.8 CLOSE

The CLOSE command is used to close the current connection.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

CLOSE			
-------	--	--	--

5.28.9 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.29 JMSXMLBindService

Service Name: **JMSXMLBindService**

The JMSXMLBindService allows XML data to be sent and received using enterprise messaging systems such as ActiveMQ, SonicMQ, TibcoMQ and WebSphereMQ.

This service uses the Java Message Service JMS 1.1 specification.

Refer to <http://www.oracle.com/technetwork/java/jms/index.html>.

If you're using:

- ActiveMQ, then you require the ActiveMQ JMS jar files.
- SonicMQ, then you require the Sonic JMS jar files.
- TibcoMQ, then you require the Tibco JMS jar files.
- WebSphereMQ, then you require the IBM WebSphereMQ JMS jar files.

You will need to know how to use the enterprise messaging system selected for use with this JMSXMLBindService. The setting up and configuring of either messaging system is beyond the scope of this guide.

The JMSXMLBindService supports the following commands:

5.29.1 SERVICE_LOAD

5.29.2 SERVICE_GET

5.29.3 CONNECT

5.29.4 SEND

5.29.5 RECEIVE

5.29.6 BIND

5.29.7 GET

5.29.8 SET

5.29.9 WRITE

5.29.10 ARCHIVE

5.29.11 COMMIT

5.29.12 ROLLBACK

5.29.13 CLOSE

5.29.14 SERVICE_UNLOAD

5.29.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	SCHEMA		Optional. See SCHEMA .
	VALIDATING		Optional. See VALIDATING .
	DOMSET		Optional. See DOMSET .

5.29.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.29.3 CONNECT

The CONNECT command is used to open a connection to the enterprise messaging system.

The currently available vendor implementations are ActiveMQ, SonicMQ, TibcoMQ and WebSphereMQ.

It is possible to create more vendor implementations by writing a Java class that implements the JMSVendor interface.

The JMSXMLBindService does a service properties lookup using the VENDOR keyword value to determine which Java class will be used to create the connection object.

```
vendor.activemq=com.lansa.service.JMSVendorActiveMQ
vendor.sonicmq=com.lansa.service.JMSVendorSonicMQ
vendor.tibcomq=com.lansa.service.JMSVendorTibcoMQ
vendor.webspheremq=com.lansa.service.JMSVendorMQSeries
```

Keyword values can also be passed to the CONNECT command by using a working list argument. The first column supplies the keyword name and the second column the keyword value.

The command keyword takes precedence over the working list keyword value entry.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword.
CONNECT	VENDOR	value	Mandatory. Message System Vendor.
	HOST	value	Mandatory. hostname hostname:port.
	PORT	value	Optional. Default. 61616 for ActiveMQ. Default. 2506 for Soni

		Default. 7222 for Tibco Default. 1414 for WebSphereMQ.
USER	value	Optional.
PASSWORD	value	Optional.
CLIENTID	value	Optional.
QUEUE	value	Mandatory.
REPLYTO	*TEMPORARY	Optional. Create and use temporary reply queue
CHANNEL	value	Conditional. Required WebSphereMQ connection
QUEUE- MANAGER	value	Optional. Used by WebSphere connection
CIPHER-SUITE	value	Optional. Used by WebSphereMQ connection
MESSAGE- PRIORITY	value	0 (lowest) to 9 (highest) Default. 4.
MESSAGE- DELIVERY	*PERSISTENT *NONPERSISTENT	Default. *PERSISTENT
MESSAGE-TIME- TO-LIVE	value	Default. 0 milliseconds A value of 0 means keep forever.
SESSION- TRANSACTION	*YES *NO	Default. *NO.
SESSION- ACKNOWLEDGE	*AUTO *CLIENT *DUPOK	Default. *AUTO.

Example

```
CONNECT VENDOR(WEBSPHEREMQ) HOST(LOCALHOST) CHANNEL  
MANAGER(USERAGENT.QUEUE.MANAGER) QUEUE(USERAGENT.QI
```

```
CONNECT VENDOR(ACTIVEMQ) HOST(LOCALHOST) QUEUE(QUEUI
```

5.29.4 SEND

The SEND command is used to an XML message. If no XML data has been created then an empty message is sent.

To include JMS message properties with the message use the optional working list argument.

The first column is the property type, the second column is the property name and the third column is the property value. Property types are I for integer, B for Boolean, S for string, H for short, L for long, D for decimal, F for float and J for special JMS properties.

The following type J properties use individual method calls on the JMS message object instead of the generic message.setStringProperty, message.setIntProperty and message.setBooleanProperty method calls.

- JMSTYPE (message.setJMSType)
- JMSMESSAGEID (message.setJMSMessageID)
- JMSCORRELATIONID (message.JMSCorrelationID)
- JMSPRIORITY (message.setJMSPriority)
- JMSREDELIVERED (message.setJMSRedelivered)
- JMSEXPIRATION (message.setJSMExpiration)
- JMSTIMESTAMP (message.setJMSTimestamp)
- JMSDELIVERYMODE (message.setJMSDeliveryMode)
- JMSREPLYTO (message.setJMSReplyTo)
- JMSDESTINATION (message.setJMSDestination)

The message identifier (MessageID) is returned in the JSMMSG field.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	MESSAGE	*BINARY	Message type. JMS BytesMessage.
		*TEXT	Default. JMS TextMessage.
	ENCODING	value.	Conditional. Used if message type is *BINARY

ARCHIVE value Default. UTF-8.
Optional. Archive file.

Example

```
SEND MESSAGE(*TEXT) SERVICE_LIST(PRPTYP,PRPNME,PRPVAL)
```

```
SEND MESSAGE(*BINARY) SERVICE_LIST(PRPTYP,PRPNME,PRPVAL)
```

5.29.5 RECEIVE

The RECEIVE command is used to receive a message.

To receive JMS message properties from the message use the optional working list argument.

The first column receives the property type, the second column receives the property name and the third column receives the property value. The property type is always S (String) for user-defined properties and J for special JMS properties.

The message identifier (MessageID) is returned in the JSMMSG field.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	ENCODING	value	Optional. See ENCODING . Default is to auto-detect encoding.
	WAITTIME	value.	Optional. Wait time in milliseconds. Default. 0 forever.
	SELECTOR	value	Optional. Message selector.
	ARCHIVE	value	Optional. Archive file.

Example

```
RECEIVE SERVICE_LIST(PRPTY,PRPNME,PRPVAL)
```

5.29.6 BIND

The BIND command binds the specified service code to the loaded inbound document or creates a new empty outbound document.

The SERVICE keyword value is used in a resource property lookup using the property key service.'value' and service.archive.'value' to locate the binding class and the binding jar file that contains the binding class and supporting classes generated by the XMLBinding Wizard. If the service binding archive entry does not exist then the binding jar file must exist in the JSM jar subdirectory instead of a user specified directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
BIND	SERVICE	value	Service name.
	TYPE	*INBOUND	Conditional. Inbound binding.
		*OUTBOUND	Conditional. Outbound binding.
	BINDTRACE	*YES	Optional. Trace the inbound bind result.
		*NO	Default.
FILTER		Optional. See FILTER .	

5.29.7 GET

The GET command is used to read the inbound document object.

Fragments use only fields, so the GET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the GET LIST command requires a working list argument.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	INSTRUCTION	value	Optional. Get XML processing instruction.

5.29.8 SET

The SET command is used to populate the outbound document object.

Fragments use only fields, so the SET FRAGMENT command requires the SERVICE_EXCHANGE(*FIELD) keyword.

Lists use only a working list, so the SET LIST command requires a working list argument.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	LIST	value	Conditional. Bind list.
	FRAGMENT	value	Conditional. Bind fragment.
	INSTRUCTION	value	Optional. Set XML processing instruction.
	DATA	value	Conditional. PI data.

5.29.9 WRITE

The WRITE command serializes the document object in preparation for the SEND command.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
WRITE	INDENT	*YES	Optional. Indent XML content.
		*NO	Default.
	INDENT-AMOUNT	value	Optional. Indent XML amount. Default. 0. If the amount is not zero then indent is enabled.
	DOCTYPE	value	Optional. Include XML <code><!DOCTYPE></code> element.
	PUBLIC	value	Conditional. The optional public component of the XML DOCTYPE declaration.
	OMIT-DECLARATION	*YES	Optional. Omit the XML declaration.
		*NO	Default.
	BINDTRACE	*YES	Optional. Trace the outbound bind result.
		*NO	Default.
	FILTER		Optional. See FILTER .

5.29.10 ARCHIVE

The ARCHIVE command saves the selected source to the archive file using the specified encoding.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
ARCHIVE	FILE	value	Mandatory. File path.
	ENCODING		Optional. See ENCODING .
	SOURCE	*RECEIVE	Optional. Archive receive source. Default. *RECEIVE.
		*SEND	Archive send source.

5.29.11 COMMIT

The COMMIT command is used to commit the current transacted session. Use the CONNECT command SESSION-TRANSACTIONED keyword to start a transacted session.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

COMMIT			
--------	--	--	--

5.29.12 ROLLBACK

The ROLLBACK command is used to rollback the current transacted session. Use the CONNECT command SESSION-TRANSACTIONED keyword to start a transacted session.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

ROLLBACK			
----------	--	--	--

5.29.13 CLOSE

The CLOSE command is used to close the current connection and binding.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

CLOSE			
-------	--	--	--

5.29.14 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.30 JSONBindFileService

The JSONBindFileService is designed to make the job of reading JSON data into applications and the creation of JSON data by applications simple. This is one of several services that are designed to consume the classes generated from the JSON Binding Wizard (a feature of the LANSAs Integrator Studio), so the use of LANSAs Integrator Studio is a key part of the JSONBindFileService.

So, for example, if you receive an order in JSON format into a directory on your server, the JSONBindFileService could be used to read the order data, convert the data to fields inside your program, from which the program can then process the data as required. Alternatively, your application might be the one that needs to create an order to be sent to a supplier. In which case your application would prepare the data required for the order then to use the JSONBindFileService to create the JSON file.

Related Services

The JSONBindFileService is one of several services that can consume the classes generated from the JSON Binding Wizard. Another service in this family combine the JSON binding with specific transport support:

- HTTPInboundJSONBindService
- HTTPOutboundJSONBindService

The JSONBindFileService does not provide transport of the file. If you need to transport the document, you may need to use one the service listed above or combine it with one of the transport focused services such as:

- FTPService - this service allows an application to send and receive files to and from a remote FTP server.
- JMSFileService - this service could be used to send and receive files using enterprise messaging systems such as ActiveMQ, SonicMQ, TibcoMQ and WebSphereMQ.

Technical Specifications

The JSONBindFileService needs to be used in conjunction with the bindings created using the JSON Binding Wizard of the LANSAs Integrator Studio. Once created you will need to move the relevant .jar and property files to the server before you can run the application.

Please refer to the [JSON Binding Wizard](#) if you are not familiar with it.

5.30.1 What can I use the JSONBindFileService for?

Pass Data Between Internal Systems

Many companies face the problem of how to integrate data from disparate systems. They might be running Oracle financials on a Windows server, and have a home grown RPG or LANSAs system running on an IBM i server, and need to move data between the two systems. JSON is becoming the preferred approach for the movement of such data.

To receive an order in JSON format and send a reply

You may be receiving orders from a distributor in JSON format, so you need a way to transform the data held within the document to information that is useful to your application - specifically, fields and lists (in the case of LANSAs applications), and subfiles (in the case of RPG, for example). As a first step in the exercise you would use the [JSON Binding Wizard](#), in LANSAs Integrator Studio, to bind the JSON elements to field names that are used in your application. Once you have done this you would use the JSONBindFileService in your application to read the data into application fields and lists (using the READ and GET commands). After that has been completed, your application can then do what it needs to in order to process the order into your system.

In many cases, you will need to send some form of acknowledgement back to the sender. This time, the JSONBindFileService can be used to create a file as a response, specifically using the WRITE and SET commands that come with this service.

How would you handle a situation where you have received many JSON documents into a directory? You could use the LIST command that comes with the JSONBindFileService. This will supply you with a list of all the files that are in the directory, and you can then use that list to process the files that you want to process.

5.30.2 Using the JSONBindFileService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you must complete the same basic steps.

For example, an application that needs to READ a JSON file would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

READ

BIND

GET

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Whereas, an application that needs to WRITE to a JSON file would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

BIND

SET

WRITE

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Refer to the [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.30.3 JSONBindFileService Properties

Properties that affect the operation of the JSONBindFileService are specified in the JSONBindFileService.properties file. The properties of interest fall into two categories:

- Properties in the form `service.xxxxx=` that associate the service name that you specify in the BIND service command with the classes generated from the JSON Binding Wizard. These properties are generated by the JSON Binding Wizard when you build your project, so you should not normally have to manually enter them. You may have to deploy them to your production environment however – refer to [JSON Binding Wizard](#) for more information.
- General properties that provide default values affecting the operation of the service. These are described in [5.30.4 JSONBindFileService Commands](#).

The default properties file is relatively uninteresting and the bulk of it is taken up with messages.

5.30.4 JSONBindFileService Commands

Your application issues commands to the JSONBindFileService by passing the command strings through the Java Services Manager using the JSM_COMMAND or JSMX_COMMAND built-in function or an API for your chosen development language.

The commands that the JSONBindFileService processes are:

SERVICE_LOAD

READ

WRITE

BIND

GET

SET

DELETE

RENAME

LIST

CLOSE

SERVICE_GET

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
---->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

SERVICE The name of the service to be loaded - in this case
JSONBindFileService.

TRACE To enable tracing from the client program use the TRACE
keyword on the SERVICE_LOAD command.

The possible values for the TRACE keyword are:

- *NO
- *YES
- *ERROR

The trace option *ERROR will turn on tracing and if the

service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.

TRACE_NAME This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(JSONBIND  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jasmcommand := 'service_load service(JSONBindFileService)  
use builtin(jsmx_command) with_args(#jasmxhdl1 #jasmcommand) to_get(#jsn
```

READ

The READ command is used to read and parse the JSON data.

When the READ command is executed, the following steps occur:

1. The file content is read as UTF-8 data and parsed .

Required

READ ----- *FILE* ----- *file path* ----->

Optional

>-- *ARCHIVE* ----- *value* -----|

Keywords

FILE This keyword is used to specify the file name and path of the JSON document.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

- Absolute path

For example:

/orders/order.json

C:/orders/order.json

C:\orders\order.json

or

- Relative path.

For example, orders/order.json (note, no '/' at the start), in which case the order.json document must reside in the orders directory under the JSM Instance directory on your server.

For the IBM i you can specify:

- Absolute path.

For example:
/orders/order.json

or

- Relative path.
For example, orders/order.json (note, no '/' at the start), in which case the order.json document must reside in the orders directory under the JSM Instance directory on your server.

Note: Whatever directory structure you specify must already exist.

This keyword is mandatory.

ARCHIVE Refer to ARCHIVE for more complete information on this keyword.

This keyword is optional.

Comments / Warnings

The ARCHIVE keyword is a very useful way to store away JSON documents after they have been processed.

Examples

RDML

* Define the fields used by the JSM Commands

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)  
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)  
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

```
CHANGE FIELD(#JSMCMD) TO('READ FILE(orders/order.json) ARCHIVE  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

* Define the fields used by the JSM Commands

```
Define FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)  
Define FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)  
Define FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)  
Define FIELD(#JSMHND) TYPE(*CHAR) LENGTH(4)
```

```
#JSMCMD := 'READ FILE(orders/order.json) ARCHIVE(archive/arc_order.js  
Use BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMHND #JSMCMD) I
```

WRITE

The WRITE command serializes the JSON object out to a specified file. You will use the SET command one or more times prior to using the WRITE command to set up the data in the document.

```

                                Required
WRITE ----- FILE ----- file path ----->

                                Optional
>-- BINDTRACE ----- *NO ----->
                                *YES
>-- FILTER ----- value -----|
```

Keywords

FILE This keyword is used to specify the file name and path of the JSON document to be created.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

Absolute path.

For example:

/orders/order.json

C:/orders/order.json

C:\orders\order.json

or

Relative path.

For example, orders/order.json (note, no '/' at the start), in which case the document order.json will be placed in the orders directory under the JSM Instance directory on your

server.

For the IBM i you can specify:

Absolute path.

For example:

/orders/order.json

or

Relative path.

For example, orders/order.json (note, no '/' at the start), in which case the document order.json will be placed in the orders directory under the JSM Instance directory on your server.

Note: The directory structure must exist.

This keyword is mandatory.

BINDTRACE This keyword is used to turn on tracing for the outbound bind result.

There are two options:

***YES** - to switch tracing on.

***NO** - to switch tracing off.

This keyword is optional.

FILTER Refer to [FILTER](#) for more complete information on this keyword.

This keyword is optional.

Examples

RDML

* Define the fields used by the JSM Commands

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

```
CHANGE FIELD(#JSMCMD) TO('WRITE FILE(response/rsp_order.json)')
```

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS

RDMLX

* Define the fields used by the JSM Commands

Define FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)

Define FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)

Define FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)

Define FIELD(#JSMHND) TYPE(*CHAR) LENGTH(4)

#JSMCMD := 'WRITE FILE(response/rsp_order.json)'

Use BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMHND #JSMCMD) T

BIND

The BIND command binds the specified service code to the loaded inbound document or creates a new empty outbound document.

The service specified as a part of the command will have been created using the JSON Binding Wizard in LANSA Integrator Studio.

You will need to specify the type of bind as outbound or inbound - a READ is normally associated with an inbound type while a WRITE is normally associated with an outbound type.

Required

```
BIND ----- SERVICE ----- value ----->
```

```
>-- TYPE ----- *INBOUND ----->  
                  *OUTBOUND
```

Optional

```
>-- BINDTRACE ----- *NO ----->  
                  *YES
```

```
>-- FILTER ----- value -----|
```

Keywords

SERVICE This keyword is used to in an JSONBindFileService property lookup using service.'value' and service.archive.'value' to locate the binding class and binding jar file to be used. The value here will be the corresponding service class name that was defined in the JSON Binding Wizard.

This keyword is mandatory.

TYPE This keyword is used to specify the type of bind to be performed, and it will depend on whether you are intending to perform a READ of an existing file or a WRITE of a new file.

There are two possible values:

***INBOUND** - using this value will bind the specified service code to the inbound document loaded as a part of the preceding READ command.

***OUTBOUND** - using this value will create a new empty outbound document.

This keyword is mandatory.

BINDTRACE This keyword is used to turn on tracing for the inbound bind result.

There are two options:

***YES** - to switch tracing on.

***NO** - to switch tracing off.

This keyword is optional.

FILTER Refer to [FILTER](#) for more complete information on this keyword.

This keyword is optional.

Comments / Warnings

The position of the BIND command in the code depends on whether you are intending to read data from an existing JSON document or you are intending to create a new JSON document. If you are reading data from an existing JSON document, the code flow will look something like this:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

READ

BIND

GET (one or more times)

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Whereas, an application that needs to WRITE to a JSON document would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

BIND
SET (one or more times)
WRITE
CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Examples

RDML

* Define JSM fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)  
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)  
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Bind

```
CHANGE FIELD(#JSMCMD) TO('BIND SERVICE(INBOUNDORDER) TY  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

* Define the fields used by the JSM Commands

```
Define FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)  
Define FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)  
Define FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)  
Define FIELD(#JSMHND) TYPE(*CHAR) LENGTH(4)
```

```
#JSMCMD := 'BIND SERVICE(INBOUNDORDER) TYPE(*INBOUND) BII  
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

GET

The GET command is used to read the data from the inbound document object. The GET command is integral part of loading data from a JSON document into your program. It must be preceded by the READ and the BIND commands.

Conditional

```
GET ----- LIST ----- value ----->
```

```
>-- FRAGMENT ----- value -----|
```

Keywords

- LIST** This keyword is used to get a list from the loaded JSON document.
Refer to the Lists and Variables section below for further details on how to use this keyword.
This keyword is conditional.
- FRAGMENT** This keyword is used to get a field fragment from the loaded JSON document.
Refer to the Lists and Variables section below for further details on how to use this keyword.
This keyword is conditional.

Comments / Warnings

You can only use one keyword at a time.

There are two ways in which you can read a list from an JSON document.

1. You can either define the relevant section of the JSON elements as a list (using the JSON Binding Wizard), and use the LIST keyword in your GET command. This will retrieve the entire list in one go into a working list.
2. Alternatively, you can define the relevant section as a collection of fragments (using the JSON Binding Wizard) and then use the FRAGMENT keyword in your GET command and place this in a loop.

See the Examples for details of how to do this.

Note: Fragments and Lists

The following points are important when using GET FRAGMENT and GET LIST together.

- When you issue a GET FRAGMENT, that fragment becomes the current fragment. That is, the fragment pointer moves to the current one.
- You need to read the fragments in an order that allows all fragments to be accessed. You can only access lists and child fragments once you have positioned the fragment pointer to its parent fragment. So, in a way, fragments are a bit like branches on a tree - to access lists and fragments further down the tree you will need to position the fragment pointer to the parent branch.
- For example, when you issue a GET FRAGMENT, a GET LIST can only access lists within this current fragment. Therefore, you should read all the lists (using GET LIST) in the current fragment before you move on to another. Once you move onto another fragment, then the lists in the previous fragments will not be accessible. You may access the lists within a specific fragment in any order.
- When a fragment becomes the current fragment it is marked as used, so once you leave it, it will not become the current fragment again.

Lists and Variables

If you are using the LIST keyword to return a list from the loaded document into your program, you will need to ensure the following steps are taken in your program.

1. Define a working list that contains the fields that you are expecting from the JSON document.
2. Use the SERVICE_LIST keyword with the LIST keyword in the GET command. The service list value should include the names of the fields in your working list without the '#'. The order of the fields should be defined here as they appear in the working list.
3. In the TO_GET portion of the JSM_COMMAND Built-In Function, include the name of the working list that will hold the values returned.

If you are using the FRAGMENT keyword to return a field or group of fields, then you will need to add the SERVICE_EXCHANGE keyword with a value of '*FIELD'.

See the Examples for details on how to do this.

Examples

RDML

* Define JSM fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Define a list to hold the order line details

```
DEF_LIST NAME(#WRKLNES) FIELDS(#LINNUM #PARTNUM #PARTI
```

* Get SalesOrder Details

```
CHANGE FIELD(#JSMCMD) TO('GET FRAGMENT(SALESORDER) SER
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

* Get Lines

```
CHANGE FIELD(#JSMCMD) TO('GET LIST(LINE) SERVICE_LIST(LINN
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

* Define the fields used by the JSM Commands

```
Define FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
Define FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
Define FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
Define FIELD(#JSMHND) TYPE(*CHAR) LENGTH(4)
```

```
#JSMCMD := 'GET FRAGMENT(CUSTOMER) SERVICE_EXCHANGE(*F
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

SET

The SET command is used for two distinct operations. Use the DIR keyword to set the current working directory or alternatively use the LIST or FRAGMENT keywords to populate the outbound document object.

Once you have completed your SET commands your next command would invariably be the WRITE command.

Conditional

```
SET ----- LIST ----- value ----->
```

```
>-- FRAGMENT ----- value ----->
```

```
>-- DIR ----- value -----|
```

Keywords

- | | |
|----------|--|
| LIST | <p>This keyword is used to place a list of values into a JSON document.</p> <p>Refer to the Lists and Variables section below for further details on how to use this keyword.</p> <p>The LIST keyword does not have any relationship with the DIR keyword.</p> <p>This keyword is conditional.</p> |
| FRAGMENT | <p>This keyword is used to place a field or group of fields into a JSON document.</p> <p>Refer to the Lists and Variables section below for further details on how to use this keyword.</p> <p>The FRAGMENT keyword does not have any relationship with the DIR keyword.</p> <p>This keyword is conditional.</p> |
| DIR | <p>This keyword is used to nominate a relative or absolute path to be set as the current directory.</p> <p>The DIR keyword does not have any relationship with the</p> |

other keywords for the SET command.
This keyword is conditional.

Comments / Warnings

There are two possible ways to add a list of data when creating a JSON document.

1. The most obvious way is to use the SET command described here with the LIST keyword, then specifying the array in a working list. With this approach the section of JSON would need to be defined as a list in the JSON Binding Wizard.
2. Alternatively, the same could be achieved by using the FRAGMENT keyword inside some looping code. Each loop would add a new row to the list. With this approach the section of JSON would need to be defined as a fragment in the JSON Binding Wizard.

Whichever way you choose, you will need to carefully decide whether the relevant section of JSON is defined as a List or a Fragment.

See the Examples following how to do this.

Lists and Variables

If you are using the LIST keyword to create a list in an JSON document, you will need to ensure the following steps are taken in your program.

1. Define a working list that contains the fields that will be passed to the JSON document.
2. Use the SERVICE_LIST keyword with the LIST keyword in the SET command. The service list value should include the names of the fields in the working list without the '#'. The order of the fields should be defined here as they appear in the working list.
3. In the TO_GET portion of the JSM_COMMAND Built-In Function, include the name of the working list that will hold the values to be placed in the JSON document.

If you are using the FRAGMENT keyword to add a field or group of fields, then you will need to add the SERVICE_EXCHANGE keyword with a value of '*FIELD'. How to do this is shown in the Examples following.

Examples

RDML

* Define JSM fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Define Order Line fields

```
DEFINE FIELD(#LINNUM) TYPE(*DEC) LENGTH(003) DECIMALS(0) C
DEFINE FIELD(#PARTNUM) TYPE(*DEC) LENGTH(003) DECIMALS(0)
DEFINE FIELD(#PARTDSC) TYPE(*CHAR) LENGTH(030) COLHDG('Des
DEFINE FIELD(#PARTAMT) TYPE(*DEC) LENGTH(010) DECIMALS(2)
DEFINE FIELD(#PARTQTY) TYPE(*DEC) LENGTH(003) DECIMALS(0) (
```

* Define the list to hold the order lines

```
DEF_LIST NAME(#WRKLNES) FIELDS(#LINNUM #PARTNUM #PARTI
```

* Set customer details

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET FRAGMENT(CUSTO
```

* Set SalesOrder header details

```
CHANGE FIELD(#JSMCMD) TO('SET FRAGMENT(SALESORDER) SERV
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
CHANGE FIELD(#JSMCMD) TO('SET LIST(LINE) SERVICE_LIST(LINN
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDML

* Define the fields used by the JSM Commands

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
Define Field(#JSMMSG) Type(*CHAR) Length(256)
Define Field(#JSMCMD) Type(*CHAR) Length(256)
Define Field(#JSMHND) Type(*CHAR) Length(4)
```

```
Define FIELD(#LINENUM) TYPE(*DEC) LENGTH(003) DECIMALS(0) L/
Define FIELD(#PARTNUM) TYPE(*DEC) LENGTH(003) DECIMALS(0) L/
Define FIELD(#PARTDSC) TYPE(*CHAR) LENGTH(020) LABEL('Part De
Define FIELD(#PARTAMT) TYPE(*DEC) LENGTH(010) DECIMALS(2) L/
Define FIELD(#PARTQTY) TYPE(*DEC) LENGTH(003) DECIMALS(0) L/
```

```
Define Field(#LINSTAT) Type(*CHAR) Length(20) Label('Line Status')
Def_List Name(#RSPLINES) Fields(#LINENUM #PARTNUM #PARTDSC #
```

* Set the customer details

```
#JSMCMD := 'SET FRAGMENT(CUSTOMER) SERVICE_EXCHANGE(*F
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

* Set the order details by using SET FRAGMENT a number of times

```
Selectlist Named(#RSPLINES)
```

```
#JSMCMD := 'SET FRAGMENT(LINE) SERVICE_EXCHANGE(*FIELD)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
```

```
#JSMCMD := 'SET FRAGMENT(PART) SERVICE_EXCHANGE(*FIELD)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
Endselect
```

DELETE

This keyword can be used to delete a file from a specified directory.

```
DELETE ----- FILE ----- file path -----|
```

Keywords

FILE This keyword is used to define the file name to be deleted. The file path must be included.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

- Absolute path.
For example:
/orders/order.json
C:/orders/order.json
C:\orders\order.json

or

- Relative path.
For example, orders/order.json (note, no '/' at the start), in which case the document order.json will be deleted from the orders directory under the JSM Instance directory on your server.

For the IBM i you can specify:

- Absolute path.
For example:
/orders/order.json

or

- Relative path.
For example, orders/order.json (note, no '/' at the start), in which case the document order.json will be deleted from the orders directory under the JSM Instance directory on your server.

This keyword is mandatory.

Examples

RDML

* Define JSM fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)  
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)  
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Delete

```
CHANGE FIELD(#JSMCMD) TO("DELETE FILE(orders/order.json)")  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

* Define the fields used by the JSM Commands

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)  
Define Field(#JSMMSG) Type(*CHAR) Length(256)  
Define Field(#JSMCMD) Type(*CHAR) Length(256)  
Define Field(#JSMHND) Type(*CHAR) Length(4)
```

* Delete

```
Change Field(#JSMCMD) To("DELETE FILE(orders/order.json)")  
Use Builtin(JSMX_COMMAND) With_Args(#JSMCMD) To_Get(#JSMSTS :
```

RENAME

The RENAME command can be used to rename a specified local file to a new name.

The file specified in for the TO value must be in the same directory as the file specified in the FROM value.

Required

```
RENAME ----- FROM ----- file path ----->
```

```
>-- TO ----- file path ----->
```

Optional

```
>-- REPLACE ----- *NO -----|  
      *YES
```

Keywords

FROM This keyword specifies the name and path of the file whose name is to be changed.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

- Absolute path.

For example:

/orders/order.json

C:/orders/order.json

C:\orders\order.json

or

- Relative path.

For example, orders/order.json (note, no '/' at the start), in which case the document order.json must be a file located in

the orders directory under the JSM Instance directory on your server.

For the IBM i you can specify:

- Absolute path.
For example:
/orders/order.json

or

- Relative path.
For example, orders/order.json (note, no '/' at the start), in which case the document order.json must be a file located in the orders directory under the JSM Instance directory on your server.

This keyword is mandatory.

TO This keyword specifies the name and the path to which the file is to be changed to. The path must be the same as that specified in the FROM keyword.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

Absolute path.

For example:

/orders/order2.json

C:/orders/order2.json

C:\orders\order2.json

or

Relative path.

For example, orders/order2.json (note, no '/' at the start), in which case the document order2.json will be placed in the orders directory under the JSM Instance directory on your server.

For the IBM i you can specify:

Absolute path.

For example:
/orders/order2.json

or

Relative path.

For example, orders/order2.json (note, no '/' at the start), in which case the document order2.json will be placed in the orders directory under the JSM Instance directory on your server.

This keyword is mandatory.

REPLACE This keyword is used to specify whether you want to overwrite any existing file of the same name as specified in the TO keyword.

There are two possible values:

***YES** - specifying this value will overwrite any existing file of the same name in the specified directory..

***NO** - if this value is set then an exception will occur if a file of the same name already exists in the directory. If you capture this exception you can report it back to the user.

The default value is *NO.

This keyword is optional.

Comments / Warnings

If you wish to place the file into another directory, you might want to consider using the ARCHIVE keyword on the READ command. This keyword will move the file that you are currently READING and place it into a new specified directory with whatever name you give it.

Examples

RDML

* Define JSM fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Rename

```
CHANGE FIELD(#JSMCMD) TO("RENAME FROM(/orders/test.json) TO(/orders/test.json)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMCMD)
```

RDMLX

* Define the fields used by the JSM Commands

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
```

```
Define Field(#JSMMSG) Type(*CHAR) Length(256)
```

```
Define Field(#JSMCMD) Type(*CHAR) Length(256)
```

```
Define Field(#JSMHND) Type(*CHAR) Length(4)
```

* Rename

```
Change Field(#JSMCMD) To("RENAME FROM(/orders/test.json) TO(/orders/test.json)
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMCMD) To_Get(#JSMSTS ;
```

LIST

The LIST command will return a list of file names found within a specified directory.

This command may be useful if a number of JSON documents need to be processed. The command could be used to populate a working list with all the documents that have been placed in a specific directory, then place the READ, BIND, and GET commands with a SELECT_LIST working on this list.

```
LIST ----- DIR ----- directory path ----->
>-- EXT ----- file extension ----->
>-- SORT ----- *NONE ----->
      *NAME
      *MODIFIED
>-- REVERSE ----- *YES -----|
      *NO
```

Keywords

- DIR This keyword is used to specify the directory that is to be searched.
- It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.
- The format of the path should be as follows:
- For Windows you can specify:
- Absolute path.
For example:
/orders
C:/orders
C:\orders
- or
- Relative path.

For example, `orders` (note, no `'/'` at the start), in which case the search will be conducted in the `orders` directory under the JSM Instance directory on your server.

For the IBM i you can specify:

- Absolute path.

For example:

`/orders`

or

- Relative path.

For example, `orders` (note, no `'/'` at the start), in which case the search will be conducted in the `orders` directory under the JSM Instance directory on your server.

This keyword is optional.

EXT This keyword is used to filter the list returned based on the file extension.

The filtering match is case insensitive.

This keyword is optional.

SORT The optional sort keyword allows sorting on file name or modified date.

The default value is `*NONE`.

REVERSE The optional reverse keyword allows the sorted order to be reversed.

The default value is `*NO`.

Comments / Warnings

If you do not specify the `DIR` value, then it will automatically return a list of files from the JSM instance directory.

Lists and Variables

This keyword requires a single field working list to receive the canonical file paths. To enable this, you should do the following:

1. Define a single field working list that will hold the returned list of files. The field needs to be long enough to hold the full canonical path and file name.

2. Use the SERVICE_LIST keyword with the LIST command. The service list value should include the name of the single field defined in the working list without the '#'.
3. In the TO_GET portion of the JSM_COMMAND Built-In Function, include the name of the working list defined above.

Examples

RDML

* Define the fields used by the JSM Commands

Define Field(#JSMSTS) Type(*CHAR) Length(020)

Define Field(#JSMMSG) Type(*CHAR) Length(256)

Define Field(#JSMCMD) Type(*CHAR) Length(256)

* Define field to hold file names

DEFINE FIELD(#FILENAME) TYPE(*CHAR) LENGTH(250)

DEF_LIST NAME(#FILELSTW) FIELDS(#FILENAME) COUNTER(#LIST

#JSMCMD := 'LIST DIR(NEWORDERS) SERVICE_LIST(FILENAME) EX'
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS

RDMLX

* Define the fields used by the JSM Commands

Define Field(#JSMSTS) Type(*CHAR) Length(020)

Define Field(#JSMMSG) Type(*CHAR) Length(256)

Define Field(#JSMCMD) Type(*CHAR) Length(256)

Define Field(#JSMHND) Type(*CHAR) Length(4)

* Define field to hold file names

Define Field(#FILENAME) Type(*CHAR) Length(250)

Def_List Name(#FILELSTW) Fields(#FILENAME) Counter(#LISTCOUNT)

#JSMCMD := 'LIST DIR(NEWORDERS) SERVICE_LIST(FILENAME) EX'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get

CLOSE

This command is used to close the current bind.

```
CLOSE ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Comments / Warnings

If you are planning to work with more than one JSON documents in a single function, then it is recommended that you CLOSE the bind of each document before you BIND the next JSON document. This is not mandatory but it will free up resources.

Examples

RDML

```
* Define the fields used by the JSM Commands
Define Field(#JSMSTS) Type(*CHAR) Length(020)
Define Field(#JSMMSG) Type(*CHAR) Length(256)

USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

RDMLX

```
* Define the fields used by the JSM Commands
Define Field(#JSMSTS) Type(*CHAR) Length(020)
Define Field(#JSMMSG) Type(*CHAR) Length(256)
Define Field(#JSMHND) Type(*CHAR) Length(4)

Use Builtin(JSMX_CLOSE) With_Args(#JSMHND) To_Get(#JSMSTS #JSM
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSA Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
                *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE

- *SERVICE - read service trace file.
- *TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.messageType (an application-defined property) in the service properties file. This property might

be defined in the service properties file with a line like this (the value "html" will be returned to the application in this case):

```
com.acme.property.messageType=html
```

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service and removes the temporary directory.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 service_unload) to_get(#jsi
```

5.31 RFIDataSourceService

Service Name: **RFIDataSourceService**

The RFIDataSourceService allows the creation of data source objects, which can be sent and received as serialized objects between a remote [JVM](#) and the LANSa Integrator JVM.

The RFIDataSourceService uses the RFIDataSource and RFIDataTable classes in the jsmrfi.jar file. This file must exist in both the remote client JVM and also the host JSM JVM.

The RFIDataSourceService supports the following commands:

5.31.1 [SERVICE_LOAD](#)

5.31.2 [SERVICE_GET](#)

5.31.3 [CREATE](#)

5.31.4 [CHECK](#)

5.31.5 [DROP](#)

5.31.6 [GET](#)

5.31.7 [PUT](#)

5.31.8 [CLEAR](#)

5.31.9 [SEND](#)

5.31.10 [SERVICE_UNLOAD](#)

5.31.1 SERVICE_LOAD

When the RFIDataSourceService is loaded using the SERVICE_LOAD command a check is made for posted content. If content has been posted to the service then the CONTENT-TYPE keyword value is checked for a value of "application/x-java-serialized-object" and the received serialized stream is converted into an object. If the object is an instance of RFIDataSource it is made available to the service else no data source object is available.

An optional keyword can be used on the SERVICE_LOAD command to allow quick and easy binding of data from the received data source to the client program.

BIND (*FIELD)

If this keyword is used, then field values in the data source object are used to update the program fields.

A map object is used to map source column names to program field names.

A mandatory list argument needs to be used with the SERVICE_LOAD command. This list supplies the field name to column name map information. The list argument can have one or two fields, the first field contains the field name and the optional second field contains the column name. If the list argument contains only one field, the column names are the same as the field names. If the list argument contains two fields and the column field name is blank the column name will be the same as the field name.

This map besides controlling the field to column name mapping also controls the possible list of fields that can be included in the data source.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	BIND	*FIELD	Optional. Bind data source fields to program fields.

5.31.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.31.3 CREATE

The CREATE command is used to create a new data source object.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CREATE	DATASOURCE	value	Mandatory. The name of the datasource to be created.

Example

```
CREATE DATASOURCE (EMPLOYEE)
```

5.31.4 CHECK

The CHECK command is used to check for the existence of a data object.

If an object is available a response status of OK is returned otherwise a response status of NOT_EXIST is returned.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CHECK	OBJECT	*DATASOURCE	Mandatory. Checks if a data source object exists.
		*FIELD	Checks if a data source field exists.
		*TABLE	Checks if a data source table exists.
	NAME	value	Optional for *DATASOURCE. Mandatory for *FIELD and *TABLE. The NAME keyword is the name of the table, field or data source. The field name is the data source field name not the program field name.

Example

```
CHECK OBJECT (*DATASOURCE)
CHECK OBJECT (*DATASOURCE) NAME (PEOPLE)
CHECK OBJECT (*TABLE) NAME (SKILLS)
CHECK OBJECT (*FIELD) NAME (EMPLOYEE_ID)
```

5.31.5 DROP

The DROP command is used to remove a data object from the DataSource object.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
DROP	OBJECT	*FIELD	Mandatory. Data source field.
		*TABLE	Data source table.
	NAME	value	Mandatory. The name of the object to be dropped.

Example

```
DROP OBJECT (*TABLE) NAME (SKILLS)
DROP OBJECT (*FIELD) NAME (EMPLOYEE_ID) –
single data source field name
```

5.31.6 GET

The GET command is used to get data from the DataSource object.

If a PROPERTY value does not exist an empty string is returned.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	OBJECT	*DATASOURCE	Mandatory. The OBJECT (*DATASOURCE) returns the data source name in the response message field.
		*FIELD	The OBJECT (*FIELD) is used to update field values from the data source. If no NAME keyword is present, then a list of field names from the list argument are used. If the NAME (*ALL) keyword is used then all the fields in the data source are used. A single field name can also be used to select one field. The field name is the program field name not the data source field name.
		*TABLE	The OBJECT (*TABLE) requires the NAME keyword to specify which data source table will be used to fill the list object argument. The map object is used to map column names to field names and if the resolved field exists in the list object its value is updated.
	NAME	value	Mandatory for *FIELD and

	*TABLE	The NAME keyword is the name of the table or field in the data source. The field name is the program field name not the data source field name.
	*ALL	All the fields from the data source.
PROPERTY	value	Optional. HTTP protocol property.

Example

GET OBJECT (*DATASOURCE)
 GET OBJECT (*TABLE) NAME (SKILLS)
 GET OBJECT (*FIELD) – use program field names from list argument
 GET OBJECT (*FIELD) NAME (EMPNO) – single program field name
 GET OBJECT (*FIELD) NAME (*ALL) – use all data source fields

5.31.7 PUT

The PUT command is used to add or update field and table data in the data source.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
PUT	OBJECT	*FIELD	Mandatory. Data source field.
		*TABLE	Data source table.
	NAME	value	Optional. The OBJECT (*TABLE) requires the NAME keyword to create a new RFIDDataTable object using the contents of the supplied list object. List field names are mapped to column names. The OBJECT (*FIELD) does not require the name keyword, if no name keyword is present, then all field names in the list argument are used. The field name is the program field name not the data source field name.

Example

```
PUT OBJECT (*TABLE) NAME (SKILLS)
PUT OBJECT (*FIELD) – all program field names from list argument
PUT OBJECT (*FIELD) NAME (EMPNO) – single program field nam
```

5.31.8 CLEAR

The CLEAR command removes the specified object from the data source.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CLEAR	OBJECT	*TABLE	Mandatory. Removes all tables from the data source.
		*FIELD	Removes all fields from the data source.

Example

CLEAR OBJECT (*TABLE) – clear all tables from data source

CLEAR OBJECT (*FIELD) – clear all fields from data source

5.31.9 SEND

The SEND command is used to serialize the current data source object and create a response for the remote JVM client program.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	OBJECT	*DATASOURCE	Mandatory. The object to be sent.

Example

```
SEND OBJECT (*DATASOURCE)
```

5.31.10 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

DO NOT use the SERVICE_UNLOAD command when sending a HTTP response. Use the SEND command to create the response and then use the JSM_CLOSE BIF to close the connection. The byte array response returned to JSM Client program by the SEND command will be become the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.31.11 RFIDDataSourceService Examples

Go to [RFIDDataSourceService Example](#) for an RDML code example.

5.32 ZipService

The ZipService provides support for:

- Creating zip file archives
- Adding files and directories to zip file archives
- Extracting files and directories from zip file archives

Archives are files that contain other files, typically in a compressed format. Zip files are the most common archive format and provide compression and file grouping.

The Zip format is widely used and supported. Most computer users will be able to create or process zip file archives. The ZipService provides an easy method for your applications to create and process these files.

Related Services

The ZipService is not dependent on other services. It can be used on its own to create, add to or extract from zip file archives. However, depending on the requirements of your application, you may wish, for example, to use one or more of the transport services to send or receive zip file archives via FTP, HTTP or email.

Note that the SMTPMailService has a feature that can collect a group of attachments into a single zip archive to attach to an out-going email message. For out-going mail applications, this feature could make it unnecessary to explicitly invoke the ZipService.

Technical Specifications

- The ZipService makes use of standard Java classes to implement its zip functionality.
- The degree of compression achieved for individual files will vary significantly depending on the nature of the file content. Binary files such as program executable files will yield relatively little compression, while text files will often achieve high compression. Some files, particularly those file formats that are already compressed (such as JPEG image files) may not compress at all or may even yield a larger file than the original following compression.

5.32.1 What can I use the ZipService for?

Many files available on the Internet and exchanged by email are distributed as zip file archives. Archives make it easy to group files and make transporting and copying those files faster. Because the zip file archive format is widely used, you can be assured that almost all recipients will be able to read and extract the contents.

On the down side, the files contained in an archive are not as readily available to the applications that may wish to use them, although modern operating systems such as Microsoft Windows and tools often provide built-in file system support for zip file archives.

Following are two examples of typical uses of zip file archives.

Exchanging files with associates and partners

There are more technologies today than ever for electronically exchanging business information with trading partners. Many, like EDI and web services, are well supported by accepted standards. However, for some requirements the complexity of these solutions is simply not justified.

A subsidiary company, for example, may have to report financial results to its parent on a regular basis. The information is contained in Excel files that are produced as part of the month-end processing for their finance application. Once all the required files have been produced, the application might invoke the ZipService to collect and compress the files into a single zip archive. The application could then invoke the SMTPMailService to send the zip archive to the recipient in the parent company as an email attachment.

Archiving infrequently used files

Often files are important even though they may be infrequently used. You might archive such files to a zip file either to save disk space or to facilitate transfer to other media for secure retention. When needed, you can extract the files from the archive again.

For some applications (especially for personal use) such archiving might be a manual process, probably using your favored graphical zip program.

In other applications you might wish to archive certain data according to defined retention requirements. A customer service application, for example, might have a facility to store documents related to a specific transaction. Very likely the documents are stored in a pre-defined directory structure that might include folders for years and months. A part of your month-end processing

might be to calculate the names of the folders containing the documents for two years ago and to use the ZipService to archive those folders and files before removing them from the system.

5.32.2 Using the ZipService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application that creates a zip archive file from one or more files in a single folder or directory would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

CREATE

ADD

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Refer to [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.32.3 ZipService Commands

Your application issues commands to the ZipService by passing the command strings through the Java Service Manager using the JSM_COMMAND or JSMX_COMMAND Built-In Function or an API for your chosen development language.

The commands that the ZipService processes are:

SERVICE_LOAD

CREATE

OPEN

ADD

GET

CLOSE

DELETE

SERVICE_GET

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
--->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

SERVICE The name of the service to be loaded - in this case ZipService.

TRACE To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.

The possible values for the TRACE keyword are:

- *NO
- *YES
- *ERROR

The trace option *ERROR will turn on tracing and if the service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.

TRACE_NAME The optional TRACE_NAME keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(ZIPSERVIC  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jasmcommand := 'service_load service(zipservice)'  
use builtin(jsmx_command) with_args(#jasmxhdl1 #jasmcommand) to_get(#jsn
```

CREATE

The CREATE command creates an archive file and makes it the current archive for this instance of the ZipService. The current archive is the archive upon which subsequent ADD and GET commands operate.

If the specified file path for the new archive already exists, then it will be replaced by the CREATE command.

```
CREATE ----- FILE ----- file path -----|
```

Keywords

FILE Specifies the file path of the archive to be created.

The file path consists of:

Path to the file (optional). If you do not specify the path, or if you specify a relative path, the file will be created relative to the JSM instance directory. The path name must conform to the naming rules of the target file system.

File name (required). The file name must conform to the naming rules of the target file system.

File extension (optional). The file extension should be .zip to ensure the file can be correctly recognized and processed by other applications.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('CREATE FILE(MYARCH
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 'create file(myarchive.zip)')
```

OPEN

The OPEN command opens an existing zip file and makes it the current archive. The current archive is the archive upon which subsequent ADD and GET commands operate.

Required

OPEN ---- *FILE* ----- *file path* ----->

Optional

>-- *MODE* ----- **READ* ----->
 **WRITE*

>-- *REPLACE* ----- **NO* -----|
 **YES*

Keywords

FILE This keyword is used to specify the file name and path of the Zip file.

It is recommended to use the forward slash as the path separator and to avoid the use of the DOS drive designator.

The format of the path should be as follows:

For Windows you can specify:

- Absolute path
For example:
/orders/order.zip
C:/orders/order.zip
C:\orders\order.zip

or

- Relative path.
For example, orders/order.zip (note, no '/' at the start), in which case the order.zip file must reside in the orders

directory under the JSM Instance directory on your server.

For the IBM i you can specify:

- Absolute path.

For example:

/orders/order.zip

or

- Relative path.

For example, orders/order.zip (note, no '/' at the start), in which case the order.zip file must reside in the orders directory under the JSM Instance directory on your server.

Note: Whatever directory structure you specify must already exist.

This keyword is mandatory.

MODE The MODE indicates whether the file is to be opened for reading or writing. Open the file for reading when you open an existing archive to interrogate its contents or to expand files contained in it. Open a file for writing when opening a file to add to or replace its contents.

The default value of *READ indicates the file is opened for reading only.

A value of *WRITE opens the file for writing. This allows new files to be added to the archive. If the file is opened for writing and it does not exist a new archive is created.

REPLACE The REPLACE keyword is used in combination with MODE(*WRITE) to indicate whether the contents of the existing file can be replaced.

The default value *NO indicates that the contents of the archive are not cleared when the file is opened. New files can be added to the archive.

A value of *YES indicates that the existing contents of the archive are cleared when the file is opened. New files can be added to the empty archive.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('OPEN FILE(MYARCHIV
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'open file(myarchive.zip)')
```

ADD

The ADD command adds files, directories and child directories to the current archive (the archive last successfully opened using the CREATE or OPEN command) in the current instance of the ZipService.

Either the PATH or FILE keyword must be specified. If both keywords are specified only the PATH keyword will be processed.

Specify the folders or files to be added using either the PATH or FILE keywords. For each of these keywords you can specify a single folder or file, or you can provide a list of entries to be added.

Conditional

```
ADD ----- FILE ----- file path ----->
                *LIST
```

```
>-- PATH ----- file path ----->
                *LIST
```

Optional

```
>-- BASE ----- *NONE ----->
                *CURRENT
                *PARENT
                path
```

```
>-- TREE ----- *YES -----|
                *NO
```

Keywords

FILE Specifies the path for one or more files to be added to the current archive. The file paths specified must refer to an existing file. Adding directories using this keyword is not supported. You can specify this keyword as either:

File path specifies the name, and optionally the path, a single directory or file to be added to the current archive. The file path consists of the following parts:

*LIST specifies a working list is provided with the command. This list includes the files, optionally with path information, to be added to the current archive. The working list must include one field to contain the path of the files to be added.

Specify either the PATH or FILE keywords. If you specify both only the PATH keyword will be processed.

PATH Specifies the paths for one or more directories or files to be added to the current archive. You can specify this keyword in one of two ways:
File path specifies the path of a single directory or file to be added to the current archive.

*LIST specifies a working list is provided with the command. This list includes the paths of one or more directories or files to be added to the current archive.

The working list can have one or two fields. The first field is expected to contain the paths of the directories or files. An optional second field can be used to specify the BASE value (refer to <BASE>) that applies to the directory or file identified by the list entry. If the second field is not provided, or if it is provided but is blank, *NONE is assumed for the BASE value.

If you specify relative paths the ZipService will look for the directories or files relative to the JSM instance directory.

Specify either the PATH or FILE keywords. If you specify both only the PATH keyword will be processed.

BASE The BASE keyword is used in conjunction with the PATH keyword. The BASE keyword controls how the path information is stored for each entry in the zip archive.

By specifying an appropriate value for this keyword you can avoid the full path being stored for each zip entry – instead you can specify that the stored path should be relative to a common root path. This is useful when the contents of the zip archive are to be extracted on a system that has a different directory structure to the source machine. If you populate the zip archive with more than one ADD command, you should take care to specify consistent and compatible values on each instance of the ADD command so as to ensure a consistent path structure in the resulting zip archive.

The possible values for the BASE keyword are:

The default value *NONE indicates the paths stored in the zip archive are not modified

*CURRENT indicates the path stored in the zip file is relative to the directory containing the entries being added to the zip file.

*PARENT indicates the path stored in the zip file is relative to the parent of the directory containing the entries being added to the zip file.

Nominate a path to be used as the relative path for all files in the zip archive.

TREE The TREE keyword is used in conjunction with the PATH keyword. The TREE keyword specifies whether the add operations should recursively traverse sub-directories of the directory(s) specified.

The possible values for the TREE keyword are:

*YES (default) the child directories of the specified directory(s) and their contents will be added to the current archive. If the child directories themselves contain further directories, those directories and their contents are also added, and so on until all the descendants of the specified directory(s) are added.

*NO child directories and their contents are not added.

Examples

RDML

This example adds a single file to the current archive. No path is specified so the ZipService will look for the file in the JSM instance directory.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('ADD FILE(MYBIGTEXT
```

RDMLX

This example adds all files in the specified path to the current archive with path information stored relative to the directory specified in the PATH keyword.

```
use builtin(jsmx_command) with_args(#jsmxhdx1 'add path(/send/orderdirect
```

GET

The GET command provides access to the contents of the current archive (the archive last successfully opened with the CREATE or OPEN commands).

Based on the ENTRY keyword value the GET command can be used to:

- Populate a working list of entries in the current archive
- Extract and uncompress entries to a specified location

Either the PATH or FILE keyword must be specified. If both keywords are specified only the PATH keyword will be processed.

Required

```
GET ----- ENTRY ----- *READ ----->  
                        *LIST
```

Optional

```
>-- PATH ----- *ALL ----->  
      file path
```

```
>-- FILE ----- file path ----->
```

```
>-- BASE ----- *NONE ----->  
                        *CURRENT  
                        path
```

```
>-- OCCURRENCE----- *ALL ----->  
                        *FIRST
```

```
>-- TO ----- path -----|
```

Keywords

ENTRY

Specifies the type of operation performed by the GET command. The possible values are:

The default value *READ extracts zip file entries matching the PATH or FILE keyword values, or extracts all entries if

neither PATH or FILE is specified. The files are extracted to the folder specified by the TO keyword. The BASE keyword controls whether path information in the zip file entries is used when extracting the files.

A value of *LIST populates the supplied working list with a list of all the zip file entries. The list must have a single field that will contain the full path of the zip file entry (as stored in the zip file).

- PATH** The PATH keyword only applies when ENTRY(*READ) is specified – that is, when the GET command is being used to extract files from the current zip archive.
- The PATH value is compared against the full path of the zip file entries to select those that are to be extracted.
- Specify either the PATH or FILE keywords. If you specify both only the PATH keyword will be processed.
- FILE** The FILE keyword only applies when ENTRY(*READ) is specified – that is, when the GET command is being used to extract files from the current zip archive.
- The FILE keyword specifies the full path of a single entry in the zip archive that is to be extracted.
- Specify either the PATH or FILE keywords. If you specify both only the PATH keyword will be processed.
- BASE** The BASE keyword only applies with ENTRY(*READ) is specified – that is, when the GET command is being used to extract files from the current zip archive.
- The BASE keyword specifies whether path information in the zip file entries is used when extracting the files to the directory specified by the TO keyword.
- The possible values for the BASE keyword are:
- The default value *NONE indicates the full paths stored in the zip file entries are used when extracting files to the directory specified by the TO keyword. The directories present in the full path stored in the zip file entry become descendant directories of the directory specified by the TO keyword.

A value of BASE(*NONE) will fail if the path information in the archive cannot be combined with the path in the TO keyword to form a valid composite path.

*CURRENT indicates the path stored in the zip file entry is disregarded and all files are extracted to the directory specified by the TO keyword.

A specific path value indicates the paths stored in the zip file entries are partially used when extracting files to the directory specified by the TO keyword. The directories that are used are those in that part of the zip file entry paths that are relative to the path specified by this keyword value. The path specified here should therefore be a path that is common to one or more of the zip file entries.

OCCURRENCE The OCCURRENCE keyword only applies when ENTRY(*READ) and FILE keyword are specified. By default all occurrences of a file are written out. If you only require the first or only occurrence of the file to be written out, then use the keyword value *FIRST. Using an occurrence value of *FIRST improves the access time to read a single file from the archive.

TO The TO keyword only applies when ENTRY(*READ) is specified – that is, when the GET command is being used to extract files from the current zip archive.

The TO keyword specifies the path to the directory that will receive the extracted files. The BASE keyword controls whether path information in the zip file entries is used when extracting the files to the directory specified by the TO keyword.

Examples

RDML

This example extracts the entire contents of the current archive to the folder /EXTRACT:

```
CHANGE FIELD(#JSMCMD) TO('GET ENTRY(*READ) TO(/EXTRACT)')  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

This example extracts the entire contents of the current archive to the folder specified by #edtExtractFolder.Value:

```
#jsmcmd := 'get entry(*read) to(' + #edtExtractFolder.Value + ') base(' + #cboE  
use builtin(jsmx_command) with_args(#jsmxhdl1 #jsmcmd) to_get(#jsmsts #
```

CLOSE

The CLOSE command closes the current archive. Once the archive is closed you may either unload the service or continue processing by creating or opening another archive using the CREATE or OPEN service commands.

```
CLOSE ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(CLOSE) TO_GET(#JSMS'
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhde1 close) to_get(#jsmxsts #jsn
```

DELETE

Deletes the specified file. If you do not specify the path, or if you specify a relative path, the ZipService attempts to delete the file relative to the JSM instance directory.

```
DELETE ---- FILE ----- file path -----|
```

Keywords

FILE The path and name of the file to be deleted.

For example:

<document name>.zip deletes a file under the JSM instance root directory

documents/<document name>.zip deletes a file under the JSM instance documents subdirectory

/<document name>.zip deletes a file under the file system root directory.

Examples

RDML

```
USE BUILTIN(TCONCAT) WITH_ARGS('DELETE FILE(' #FILEPATH ')') ;  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 'delete file(deleteme.zip)') t
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSa Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
        *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE

- *SERVICE - read service trace file.
- *TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.messageType (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html")

will be returned to the application in this case):

```
com.acme.property.messageType=html
```

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service, closing any input or output streams and removing temporary directories or files.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TC
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 service_unload) to_get(#jsi
```

5.32.4 ZipService Examples

Examples are supplied for the most common client programming languages:

[RDML](#)

[RDMLX](#)

[RPG](#)

5.33 PDFSpoolFileService

The PDFSpoolFileService allows IBM i spool files to be converted to PDF documents. Use this service to create a PDF file from a single IBM i spool file. If the PDF file already has information this will be replaced with the spool file information.

The Adobe Portable Document Format (PDF) is widely used as a portable and well-recognised document format. Most users will be able to process PDF files and the format preserves both content and formatting.

Using this service you can write applications that automatically convert spool files (for example from scheduled jobs) to PDF files, for example for archiving or to send to nominated recipients via email (perhaps using the SMTPMailService).

The service provides commands for connecting to an IBM i host server, retrieving the contents of a nominated output queue and creating a PDF document for a nominated IBM i spool file. Each page in the spool file becomes one page in the resulting PDF document and a range of keywords can be specified to format the PDF document.

Related Services

The PDFSpoolFileService is not dependent on other services. It can be used on its own to create PDF files from an IBM i spool file or delete PDF files.

However, depending on the requirements of your application, you may wish, for example, to use one or more of the transport services to send PDF files via FTP, HTTP or email.

Note that the SMTPMailService has a feature that can collect a group of attachments into a single zip archive to attach to an outgoing email message.

Technical Specifications

- The PDFSpoolFileService is only relevant when connecting to an IBM i JSM Server.
- This service is only suitable for SCS or AFPDS spool files.
- This service requires the [IBM Toolbox for Java](#) to be installed.
- This service requires i5/OS 5761SS1 Option 43 Additional Fonts licensed program to be installed.
- This service uses the open source iText API classes. Refer to www.lowagie.com/iText.

5.33.1 What can I use the PDFSpoolFileService for?

Many files available on the Internet and exchanged by email are distributed as PDF files. Because the PDF file format is widely used, you can be assured that nearly all recipients will be able to read and extract the contents, or have access to download the free PDF reader software.

The following example when you may use conversion of an IBM i spool file to a PDF file.

Simplified distribution of application reports

Replace the need for distribution of hardcopy reports by converting your IBM i reports to a PDF format and distributing by email to the appropriate parties.

For example, you may have an IBM i RPG or LANSAP application generating a monthly sales report which is to be distributed to the regional sales managers. Instead of printing a copy for each manager and distributing by conventional mail services (i.e. mail or courier) you could convert the report to a PDF format and then attach this PDF document to an email for distribution to the group of sales managers. Implementing this process will save time and substantially reduce your printing and delivery costs.

5.33.2 Using the PDFSpoolFileService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application that creates a PDF file from an IBM i spool file would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

CONNECT

LIST

CREATE

DISCONNECT

SERVICE_UNLOAD

SERVICE_UNLOAD

JSM(X)_CLOSE

Refer to [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.33.3 PDFSpoolFileService Commands

Your application issues commands to the PDFSpoolFileService by passing the command strings through the Java Service Manager using the JSM_COMMAND or JSMX_COMMAND Built-In Function, or an equivalent Built-In Function or API for your chosen development language.

The commands that the PDFSpoolFileService processes are:

SERVICE_LOAD

CONNECT

LIST

SET

CREATE

DISCONNECT

DELETE

SERVICE_GET

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
--->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

SERVICE The name of the service to be loaded - in this case *PDFSpoolFileService*.

TRACE To enable tracing from the client program use this keyword on the SERVICE_LOAD command.

The possible values for the TRACE keyword are:

- *NO
- *YES
- *ERROR

The trace option *ERROR will turn on tracing and if the

service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.

TRACE_NAME This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(PDFSPPOOL  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jsmcommand := 'SERVICE_LOAD SERVICE(PDFSPPOOLFILESERVICE)  
use builtin(jsmx_command) with_args(#jsmxhdle1 #jsmcommand) to_get(#jsn
```

CONNECT

The CONNECT command connects to the IBM i host machine. If no HOST keyword is present then the connection is established using the native access method.

```
CONNECT ----- HOST ----- host:port ----->
      >-- USER ----- user profile ----->
      >-- PASSWORD ---- password -----|
```

Keywords

- HOST Nominate a host to connect to. If no HOST keyword is present then the connection is established to the same machine where the JSM Server is active.
- USER User profile to be used for connection to host.
- PASSWORD Password for supplied user.

Comments / Warnings

To use the CONNECT command without specifying a specific HOST, USER and PASSWORD, the JSM Server must be running on the IBM i where you are attempting to retrieve spool files.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('CONNECT') TO_GET(#J:
```

or

```
CHANGE FIELD(#JSMCMD) TO(" CONNECT HOST(ISERIES1) USER(U
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

In this example you explicitly OPEN the JSM Server on the IBM i where you

intend to retrieve spool files. By nominating the <host>:<port> on the JSMX_OPEN command you can then CONNECT to the same server without specifying a HOST, USER or PASSWORD.

```
* connect the JSMX client to an IBM  
i based Java Service Manager and start a thread for the service  
use builtin(jsmx_open) with_args('ISERIES1:4570') TO_GET(#jsmxsts #jsmx
```

```
* Load the service  
#jsmcommand := 'SERVICE_LOAD SERVICE(PDFSPoolFILESERVICE)'  
use builtin(jsmx_command) with_args(#jsmxhdle1 #jsmcommand) TO_GET(#
```

```
* connect to the IBM i host  
use builtin(jsmx_command) with_args(#jsmxhdle1 'Connect') TO_GET(#jsmx
```

or

If by default the JSMX_OPEN connects to a JSM Server running on a different machine than where you want to retrieve the spool files information, you will need to supply the HOST, USER and PASSWORD for the machine you want to CONNECT to.

```
* connect the JSMX client to an IBM  
i based Java Service Manager and start a thread for the service  
use builtin(jsmx_open) to_get(#jsmxsts #jsmxmsg #jsmxhdle1)
```

```
* Load the service  
#jsmcommand := 'SERVICE_LOAD SERVICE(PDFSPoolFILESERVICE)'  
use builtin(jsmx_command) with_args(#jsmxhdle1 #jsmcommand) TO_GET(#
```

```
* connect to the IBM i host  
#jsmcommand := 'Connect Host(' + #JSMServer + ') User(' + #JSMUserid + ')'  
use builtin(jsmx_command) with_args(#jsmxhdle1 #jsmcommand) TO_GET(#
```

LIST

The LIST command is used to obtain a list of spool files from the specified output queue.

Required

```
LIST ---- LIBRARY ----- library name ----->
```

```
>-- QUEUE ----- output queue ----->
```

Optional

```
>-- USER ----- *ALL ----->  
      user ID
```

```
>-- USERDATA ----- *ALL ----->  
      userdata
```

```
>-- FORMTYPE ----- *ALL -----|  
      formtype
```

Keywords

LIBRARY A valid IBM i library name must be supplied.

QUEUE A valid IBM i output queue must be supplied.

SERVICE_LIST Only required for RDML clients.

The LIST command requires a working list with six or ten fields to receive the spool file information.

The fields can be of any name and size, it is the sequence of the fields that is important. The SERVICE_LIST field sequence, size and type must match the fields defined in the DEF_LIST included in the return keywords.

NAME The name of the spool file.

NUMBER The spooled file number of the specified file.

JOBNAME The name of the job that created the spooled file.
JOBUSER The name of the user who produced the spooled file.
JOBNUMBER The number of the job in the system.
STATUS Status of spool file.
PAGES Page count.
DATE Creation date (YYYY-MM-DD).
TIME Creation time (HH:MM:SS).
USERDATA User data.

Refer to Reserved Keywords for your appropriate JSM Client for more information.

USER Indicates that only spool files with a **JOBUSER** matching this **USER** should be returned.
This should be a valid IBM i User ID or *ALL.

USERDATA Indicates that only spool files matching the **USERDATA** specified should be returned.
The default value is *ALL

FORMTYPE Indicates that only spool files matching the **FORMTYPE** specified should be returned.
This should be a valid IBM i spool file **FORMTYPE** (e.g. *STD) or *ALL.

Comments / Warnings

Use the **USER**, **USERDATA** and **FORMTYPE** filters to limit the amount of spool files returned.

Examples

RDML

Using **RDML** the **SERVICE_LIST** keyword is required to provide appropriate fields for each column in the working list to be returned.

```

DEF_LIST NAME(#SPOOLST) FIELDS(#FLENAM #FLENUM #JOBNAM
USE BUILTIN(TCONCAT) WITH_ARGS('LIST LIBRARY(' #JSMLIB ') QU

USE BUILTIN(TCONCAT) WITH_ARGS('LIST LIBRARY(' #JSMLIB ') QU

IF COND(#JSMUSER *NE *BLANK')
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #JSMUSER) TO_GET
ELSE
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD '*ALL') TO_GET(#JS
ENDIF

USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD ') USERDATA(*ALL)

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

```

RDMLX

Using RDMLX the working list does not need to be explicitly described to the JSM service (like RDML) as the LANSAs compiler automatically determines the structure of the working list nominated on the returned arguments.

```

Def_list name(#spoolst) fields(#file name #filenumber #jobname #jobuser job

#jasmcommand := 'LIST LIBRARY(QUSRSYS) QUEUE(' + #JSMOUTQ + ')

use builtin(jsmx_command) with_args(#jasmxhdl1 #jasmcommand) TO_GET(#

```

SET

The SET command is used to specify page images to be used by the CREATE command.

Required

```
SET ----- OBJECT ----- *IMAGES -----|
```

Keywords

OBJECT *IMAGES

SERVICE_LIST Only required for RDML clients.

The LIST command requires a working list with six fields to pass the image information to the service.

The fields can be of any name and size, it is the sequence of the fields that is important.

PAGE Page number.

X X location of image.

Y Y location of image.

WIDTH Width of image.

HEIGHT Height of image.

PATH Path of image file.

Refer to Reserved Keywords for your appropriate JSM Client for more information.

Comments / Warnings

To include the same image on all pages, use a page number of 0.

The unit of measurement for positioning the image is the point.

There are 72 points in 1 inch.

The X co-ordinate runs from the left side of the page to the right size of the page.

The Y co-ordinate runs from the top of the page to the bottom of the page.

The top left hand corner of the page is 0,0.

CREATE

The CREATE command reads the specified spool file and creates the PDF document.

Each spool file page will cause a new PDF page to be created. Use appropriate font size, leading and printer file options to fit the spool file page on a single PDF document page.

A spool file of 132 columns and 57 lines per page can fit on an A4 landscape page using a font size of 8 and leading of 8.5.

To accommodate the large number of keywords, a working list can be used to pass a list of keywords. Create a working list with two fields, the first field contains the keyword and the second field contains its value. Add an entry for each keyword required. Any keyword specified directly on the command will take precedence over a corresponding keyword entry in the working list. Refer to the examples to see how this works.

Required

```
CREATE ----- DOCUMENT ----- file path -----  
>
```

```
>-- APPEND ----- *NO ----->  
      *YES
```

```
>-- NAME ----- spool file name ----->
```

```
>-- JOBNAME ----- spool file job name ----->
```

```
>-- JOBUSER ----- spool file job user ----->
```

```
>-- JOBNUMBER ----- spool file job number ----->
```

Optional

```
>-- NUMBER ----- 1 ----->  
      numeric value
```

>-- *PASSWORD* ----- *password* ----->
>-- *MASTER* ----- *master* ----->
>-- *AUTHOR* ----- *author* ----->
>-- *TITLE* ----- *title* ----->
>-- *SUBJECT* ----- *subject* ----->
>-- *KEYWORDS* ----- *keywords* ----->
>-- *CREATOR* ----- *creator* ----->
>-- *ENCODING* ----- **HPT* ----->
 **CSF*
 Cp37
 Cp273
 Cp277
 Cp278
 Cp280
 Cp284
 Cp285
 Cp297
 Cp420
 Cp423
 Cp424
 Cp500
 Cp838
 Cp870
 Cp871
 Cp875
 Cp880
 Cp905
 Cp930
 Cp933
 Cp935
 Cp937
 Cp939

Cp1025
Cp1026
Cp1112
Cp1122
Cp1123
Cp1130
Cp1140
Cp1141
Cp1142
Cp1143
Cp1144
Cp1145
Cp1146
Cp1147
Cp1148
Cp1149
Cp1153
Cp1154
Cp1155
Cp1156
Cp1157
Cp1158
Cp1160
Cp1164
Cp1371
Cp1388
Cp1399
Cp4971
Cp5026
Cp5035

>-- FONT ----- *AUTO ----->

*COURIER
*COURIER2
*COURIER3
*COURIER4
*MINCHO
*GOTHIC
*SIMSUN

**DFKAISB*
**MINGLIU*
**BATANG*
**COURIERNEW*
**MTSANSWT*
**MTSANSWTIN*
**MTSANSWTJ*
**MTSANSWTJEA*
**MTSANSWTK*
**MTSANSWTKEA*
**MTSANSWTME*
**MTSANSWTTC*
**MTSANSWTTCEA*
**MTSANSWTTCTWEA*
**THRNDWT*
**THRNDWTJ*
**THRNDWTK*
**THRNDWTME*
**THRNDWTSC*
**THRNDWTTC*

>-- *FONTSIZE* ----- 10 ----->
 numeric value

>-- *LEADING* ----- (*fontsize * 1.5*) ----->
 numeric value

>-- *PAGES* ----- *n,n-n* ----->

>-- *PAGETYPE* ----- **TEXT* ----->
 **IMAGE*

>-- *PAGESIZE* ----- **A4* ----->
 **A0 -> *A10*
 **B0 -> *B5*
 **LETTER*
 **HALFLETTER*
 **LEDGER*
 **LEGAL*

**NOTE*
width, height

>-- SCALING ----- n ----->

>-- MARGIN ----- 36 points ----->
left, right, top, bottom (points)

>-- ORIENTATION -- *PORTRAIT -----|
**LANDSCAPE*

Keywords

DOCUMENT	The name of the PDF file to be created or replaced. For example: <document name>.pdf creates a PDF file under the JSM instance root directory documents/<document name>.pdf creates a PDF file under the JSM instance documents subdirectory /<document name>.pdf creates a PDF file under the file system root directory.
APPEND	Append the created spooled file pages to the existing document. The default is *NO.
NAME	Spool file name. This must be a valid spool file name on the nominated IBM i output queue.
NUMBER	Spool file number. Default value of 1 is used if a value is not assigned. This must be a valid spool file number on the nominated IBM i output queue.
JOBNAME	Spool file job name. This must be a valid job name on the nominated IBM i output queue.
JOBUSER	Spool file job user. This must be a valid job user on the nominated IBM i output queue.
JOBNUMBER	Spool file job number. This must be a valid job number on the nominated IBM i output queue.

PASSWORD	A Password can be associated with the generated PDF file to restrict the opening of the document. This Password is case sensitive.
MASTER	If a Master Password is associated with the PDF document, this password must be supplied before you can set or change security settings. This Master Password is case sensitive.
AUTHOR	A text value to indicate the Author of the PDF document.
TITLE	A text string to be used as the title of the generated PDF document.
SUBJECT	A text string to indicate what the generated PDF document is about.
KEYWORDS	One or more words can be supplied to assist with searching for a PDF document. Keywords can be looked at as categories you can use to group similar or related documents.
CREATOR	A text value to indicate who created the PDF document.
ENCODING	The encoding is used to specify automatic host print transform conversion, automatic or specified copy to stream file conversion. The default encoding is *HPT. Use *CSF for automatic copy to stream file conversion or use a codepage value to specify the copy to stream file conversion ccsid.
FONT	<p>The font used to draw the spool file text is fixed width Courier, Heisei Mincho or Heisei Kaku Gothic. The default font is *AUTO.</p> <p>The following fonts are available by default.</p> <p>*AUTO Font: Determine font using selected encoding.</p> <p>*COURIER Font: Courier</p> <p>*COURIER2 Font: Courier</p> <p>*COURIER3 Font: Courier</p> <p>*MINCHO Heisei Mincho.</p> <p>*GOTHIC Heisei Kaku Gothic.</p>

The other font support requires that i5/OS 5761SS1 Option 43 Additional Fonts licensed program to be installed.

FONTSIZE	The font size in points. The default setting is 10 point.
LEADING	Allows you to control the blank space before each line in the PDF document. The Default value is the (FONTSIZE * 1.5).
PAGES	A comma separated list of single page numbers and page ranges. The default is to select all pages.
PAGETYPE	This keyword specified the type of page generated. *IMAGE generates pages in TIFF G4 format. The SCALING keyword controls the percentage scaling of the image. The default value of *TEXT generates pages in text format. The FONT, FONTSIZE and LEADING keywords control the text output.
PAGESIZE	The page size for the generated PDF document. The default value is *A4. Valid values are: *A0 to *A10 *B0 to *B5 *LETTER *HALFLETTER *LEDGER *LEGAL *NOTE width, height
SCALING	The percentage scaling to be applied to the generated page image. The default value is 18.
MARGIN	Page margins are the blank space around the edges of the page. The default value is 36 points (all sides) but you can set your own margins by nominating left, right, top and bottom values.
ORIENTATION	The default option is *PORTRAIT. Alternately you can

nominate to generate as *LANDSCAPE.

Comments / Warnings

Ensure the user used to CONNECT to the host is authorized to work with the relevant spool files.

Examples

RDML

Particularly in RDML with its 256 field length restriction, it may be useful to use the SERVICE_LIST keyword to accommodate all the required keywords and values for the CREATE command.

Note: In this example the orientation will be *PORTRAIT as the command value will override any keywords provided in the SERVICE_LIST.

```
DEF_LIST NAME(#PDFLST) FIELDS((#PDFKEY *INP) (#PDFVALUE *IN
```

```
DEFINE FIELD(#PDFKEY) TYPE(*CHAR) LENGTH(20)  
DEFINE FIELD(#PDFVALUE) TYPE(*CHAR) LENGTH(50)
```

```
CHANGE FIELD(#PDFKEY) TO(AUTHOR)  
CHANGE FIELD(#PDFVALUE) TO('LANSA PTY LTD')  
ADD_ENTRY TO_LIST(#PDFLST)
```

```
CHANGE FIELD(#PDFKEY) TO(TITLE)  
CHANGE FIELD(#PDFVALUE) TO('ANNUAL SALES REPORT')  
ADD_ENTRY TO_LIST(#PDFLST)
```

```
CHANGE FIELD(#PDFKEY) TO(PAGESIZE)  
CHANGE FIELD(#PDFVALUE) TO(*A5)  
ADD_ENTRY TO_LIST(#PDFLST)
```

```
CHANGE FIELD(#PDFKEY) TO(ORIENTATION)  
CHANGE FIELD(#PDFVALUE) TO(*LANDSCAPE)  
ADD_ENTRY TO_LIST(#PDFLST)
```

```
CHANGE FIELD(#JSMCMD) TO('CREATE ORIENTATION(*PORTRAIT)
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jsmcommand := 'CREATE DOCUMENT(' + #JSMPDFdoc + ') NAME(' + #f  
use builtin(jsmx_command) with_args(#jsmxhdl1 #jsmcommand) to_get(#jsn
```

DISCONNECT

The DISCONNECT command closes the current connection.

DISCONNECT ----- no keywords -----|

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(DISCONNECT) TO_GET(
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 DISCONNECT) TO_GET(
```

DELETE

Deletes the specified PDF file.

If only a PDF file name is used, the command attempts to be delete this file from the root directory of the JSM Server instance.

```
DELETE ---- FILE ----- file path -----|
```

Keywords

FILE The name of the PDF file to be deleted.

For example:

<document name>.pdf deletes a PDF file under the JSM instance root directory

documents/<document name>.pdf deletes a PDF file under the JSM instance documents subdirectory

/<document name>.pdf deletes a PDF file under the file system root directory.

Comments / Warnings

The DELETE Command does not require a connection to a host IBM i as the path for the PDF file to be deleted is relative to the JSM Server Instance.

Examples

RDML

```
USE BUILTIN(TCONCAT) WITH_ARGS('delete file(' #JSMPDFDOC ')') TC  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jsmcommand := 'delete File(' + #jsmpdfdoc + )'  
use builtin(jsmx_command) with_args(#jsmxhdl1 #jsmcommand) to_get(#jsn
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSA Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
                *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE

- *SERVICE - read service trace file.
- *TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.message type (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html")

will be returned to the application in this case):

```
com.acme.property.messageType=html
```

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service and disconnects any open system.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO_;
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 SERVICE_UNLOAD) to_;
```

5.33.4 PDFSpoolFileService Examples

Examples are supplied for the most common client programming languages:

[RDML](#)

[RDMLX](#)

[RPG](#)

5.33.5 Troubleshooting

Any generated PDF document will appear in the destination path after the CREATE command has been executed. If you attempt to open this PDF file before the service is closed you will get a message "... file is unsupported or damaged.

5.34 PDFDocumentService

Service Name: **PDFDocumentService**

The PDFDocumentService allows complex PDF documents to be created.

This service uses the open source iText API classes.

iText requires Bouncy Castle for password protecting files and other encryption and certificate tasks.

Bouncy Castle requires the JCE unlimited strength policy files 'local_policy.jar' and 'US_export_policy.jar' to be installed in the JDK's lib/security directory.

For more information refer to:

[The Legion of the Bouncy Castle](#)

[iText, Programmable PDF Software](#)

[JCE Unlimited Strength Policy Files](#)

The PDFDocumentService supports the following commands:

[5.34.1 SERVICE_LOAD](#)

[5.34.2 SERVICE_GET](#)

[5.34.3 CREATE](#)

[5.34.4 ADD](#)

[5.34.5 CLOSE](#)

[5.34.6 DELETE](#)

[5.34.7 SIGN](#)

[5.34.8 FILL](#)

[5.34.9 READ](#)

[5.34.10 SERVICE_UNLOAD](#)

Also see

[5.34.11 XML Content](#)

[5.34.12 Example XML Content](#)

[5.34.13 Example RDML function](#)

5.34.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_LOAD	TIMEZONE	value	Optional. See 5.1.3 Time Zones . This overrides the timezone service property. If no service property then the default TimeZone is used.

5.34.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.34.3 CREATE

The CREATE command creates an empty document.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CREATE	DOCUMENT	value	Mandatory. PDF file.
	CONTENT	value	Mandatory. XML content file.
	PASSWORD	value	Optional. User password.
	MASTER	value	Optional. Master password
	VERSION	1.2	Optional. PDF Version.
		1.3	Default. iText default.
		1.4	
		1.5	
		1.6	
		1.7	

Example

```
CREATE DOCUMENT(report.pdf) CONTENT(report.xml)
```

5.34.4 ADD

The ADD command reads the XML content file for the specified content section and adds the content to the current document or template. LANSAs fields passed with the command can be merged into any XML attribute using the {field} notation.

The PDF document can be broken down into small content regions and each ADD command call can generate the content for that region.

When adding content, that contains a table with a height attribute, if all the rows in the working list do NOT fit into the specified table height then a status of INCOMPLETE is returned. The working list is modified and unused entries are left in the list ready for another ADD content call.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
ADD	CONTENT	value	Mandatory. Name of XML content section.
	OFFSET	*NONE	Optional. Default. The zero X and Y co-ordinate is the bottom left corner of the page.
		*MARGIN	Add the left margin value to the X co-ordinate and the bottom margin to the y co-ordinate. This will offset the location by the current margin amount.
	TEMPLATE	value	Optional. Name of template to receive content.
PASSWORD	value	Optional. Required by import PDF pages, if PDF file is password protected.	

Example

ADD CONTENT(main) SERVICE_LIST(...)

ADD CONTENT(image) TEMPLATE(logo) SERVICE_LIST(...)

Also see

[5.34.11 XML Content](#)

5.34.5 CLOSE

The CLOSE command closes the document.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

CLOSE			
-------	--	--	--

Example

CLOSE			
-------	--	--	--

5.34.6 DELETE

The DELETE command deletes the specified file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
DELETE	FILE	value	Mandatory. File Path.

Example

```
DELETE FILE (myreport.pdf)
```

5.34.7 SIGN

The SIGN command signs the specified PDF document.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SIGN	DOCUMENT	value	Mandatory. PDF file.
	CONTENT	value	Mandatory. XML content file.
	PASSWORD	value	Optional. User password.
	MASTER	value	Optional. Master password
	NAME	value	Mandatory. Name of signature.
	OUTPUT	value	Mandatory. Signed PDF file.

Example

```
SIGN DOCUMENT(report.pdf) CONTENT(signpdf.xml) NAME(signature1)
```

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<document>
```

```
  <signature name="signature1" keystore="pdfself.jks" password="password" |
```

```
</document>
```

5.34.8 FILL

The FILL command allows specified Acrobat form fields to be set. This command requires a field list to be pass, so values can be obtained. The form usage attribute can have a value of flatten to remove form field edit capability, remove to remove form usage rights, the default is to preserve form usage rights.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
FILL	DOCUMENT	value	Mandatory. PDF file.
	CONTENT	value	Mandatory. XML content file.
	PASSWORD	value	Optional. User password.
	MASTER	value	Optional. Master password
	NAME	value	Mandatory. Name of form.
	OUTPUT	value	Mandatory. Output PDF file.

Example

```
FILL DOCUMENT(form.pdf) CONTENT(form.xml) NAME(insurance) OUT  
SERVICE_EXCHANGE(*FIELD)
```

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<document>
```

```
<form name="insurance" usage="flatten">
```

```
<field name="FIRM NAME" value="{FNAME}"/>
```

```
<field name="FIRM ADDRESS" value="{FADDR}"/>
```

```
</form>
```

</document>

5.34.9 READ

The READ command reads the specified document and returns document information into the list argument. The list needs to have two fields and the fields large enough to receive the data. The first field identifies the information and the second field has the value.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
READ	DOCUMENT	value	Mandatory. PDF file.
	PASSWORD	value	Optional. User password.
	INCLUDE	*PAGESIZE	Optional. Include the page size of each page in the document.

Example

```
READ DOCUMENT(report.pdf) SERVICE_LIST(KEY,VALUE)
```

5.34.10 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.34.11 XML Content

The XML content file requires a mandatory root document element. Child content elements are used to group other XML elements. The ADD command's CONTENT keyword value refers to the name attribute value of the XML content element.

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<document>
```

```
  <content name="main">
```

```
  </content>
```

```
  <content name="logo">
```

```
  </content>
```

```
</document>
```

The unit of measurement used for positioning document content is the point. There are 72 points in 1 inch.

If you want to create a rectangle in PDF that has the size of an A4-page, you have to calculate the number of points:

$21 \text{ cm} / 2.54 = 8.2677 \text{ inch}$, $8.2677 * 72 = 595 \text{ points}$

$29.7 \text{ cm} / 2.54 = 11.6929 \text{ inch}$, $11.6929 * 72 = 842 \text{ points}$

The default border of 36 points corresponds with half an inch.

The X co-ordinate runs from the left side of the page to the right size of the page.

The Y co-ordinate runs from the bottom of the page to the top of the page.

The bottom left hand corner of the page is 0,0 and the top right would be 595, 842 for an A4 page.

All content XML attributes can have dynamic values. LANSa field values can be bound to the attribute's value at runtime by using a { } notation.

The following example shows how to use LANSAs field values as attribute values.

```
<barcode x1="20" y1="500" type="CODE128" value="{FIELD1}"/>
```

```
<barcode x1="{FIELD2}" y1="{FIELD3}" type="{FIELD4}" value="{FIELD5}"/>
```

Several Built-In Functions are also available.

- @date
- @page
- @rowcount

```
<if compare="{@page}" value="11" goto="additional"/>
```

```
<phrase value="{@date}" date-format="dd/MM/yyyy"/>
```

Element - document

The document element is the root element of all other elements and describes the PDF document.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
document	page-size	A0 to A10	Default. A4.
		B0 to B5	
		letter	
		value	
	margin	value	left, right, top, bottom Default. 36 points.
	orientation	portrait	Default.
		landscape	
	background	color	
	border	boolean	Default. false.
	grid	boolean	Default. false.
	author	string	
	title	string	
	subject	string	
	keywords	string	
creator	string	LANSAs Integrator Document Service	
hide-toolbar	boolean	Default. false.	
hide-menubar	boolean	Default. false.	
hide-windowui	boolean	Default. false.	

allow-printing	boolean	Default. true.
allow-copy	boolean	Default. true.
allow-modify-contents	boolean	Default. true.
allow-modify-annotations	boolean	Default. true.
allow-fillin	boolean	Default. true.
allow-screenreader	boolean	Default. true.
allow-assembly	boolean	Default. true.
allow-degraded-printing	boolean	Default. true.
compression	boolean	Default. false.
page-layout	onecolumn singlepage twopageleft twopageright twocolumnleft twocolumnright	Default.
fitwindow	boolean	Default. false.
centerwindow	boolean	Default. false.
display-doctype	boolean	Default. false.
printscaling-none	boolean	Default. false.
page-direction	L2R R2L	
page-mode	useoc usenone	Default.

	usethumbs	
	useoutlines	
	useattachments	
	fullscreen	
nonfullscreen-page-mode	useoc	
	usenone	Default.
	usethumbs	
	useoutlines	

The page-layout attribute sets the page layout to be used when the document is opened (choose one).

- singlepage - Display one page at a time.(default)
- onecolumn - Display the pages in one column.
- twocolumnleft - Display the pages in two columns, with odd numbered pages on the left.
- twocolumnright - Display the pages in two columns, with odd numbered pages on the right.
- twopageleft - Display the pages two at a time, with odd numbered pages on the left.
- twopageright - Display the pages two at a time, with odd numbered pages on the right.

The page-mode attribute sets the page mode to be used when the document is opened (choose one).

- useoc - Optional content group panel visible
- usenone - Neither document outline nor thumbnail images visible.(default)
- usethumbs - Thumbnail images visible.
- useoutlines - Document outline visible.
- useattachments - Attachments visible.
- fullscreen - Full-screen mode, with no menu bar, window controls, or any

other window visible.

hide-toolbar - A flag specifying whether to hide the viewer application's tool bars when the document is active.

hide-menubar - A flag specifying whether to hide the viewer application's menu bar when the document is active.

hide-windowui - A flag specifying whether to hide user interface elements in the document's window (such as scroll bars and navigation controls), leaving only the document's contents displayed.

fitwindow - A flag specifying whether to resize the document's window to fit the size of the first displayed page.

centerwindow - A flag specifying whether to position the document's window in the center of the screen.

display-doctitle - A flag specifying whether to display the document's title in the top bar.

printscaling-none - Indicates that the print dialog should reflect no page scaling.

The page-direction attribute has no direct effect on the document's contents or page numbering, but can be used to determine the relative positioning of pages when displayed side by side or printed n-up (choose one).

- L2R - Left to right
- R2L - Right to left (including vertical writing systems such as Chinese, Japanese, and Korean)

The nonfullscreen-page-mode sets the page mode when exiting full-screen mode.

It is meaningful only if the page mode is fullscreen (choose one).

- useoc - Optional content group panel visible
- usenone - Neither document outline nor thumbnail images visible
- usethumbs - Thumbnail images visible
- useoutlines - Document outline visible

Element - content

The content element is a top-level element and is a container for other elements.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
content	name	string	

Element - template

The template element is a top-level element that defines a template.

The template name is used by the ADD command TEMPLATE keyword.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
template	name	string	
	width	integer	width
	height	integer	height

Example

```
<template name="logo" width="200" height="300">  
  <image x1="0" y1="300" file="logo.jpg" scale="80"/>  
</template>
```

Element - color

The color element is a top-level element that defines a color.

Pre-defined colors are available:

- black
- blue
- cyan
- darkgray
- darkgray
- gray
- gray
- lightgray
- lightgray
- green
- magenta
- orange
- pink
- red
- white
- yellow

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
color	name	string	
	rgb	value	red, green, blue

Example

```
<color name="charcoal" rgb="100,80,90"/>
```

Element - style

The style element is a top-level element that defines a style.

The font attribute is used to select one of the built-in fonts.

The file attribute allows an external True Type Font file to be read and embedded into the PDF document.

The style attribute can contain a single value or one or more values separated by a space.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value	
style	name	string		
		font	Courier	Type 1 font.
			Courier-Bold	Base Font.
			Courier-Oblique	Base Font.
			Courier-BoldOblique	Base Font.
			Helvetica	Default. Type 1 font.
			Helvetica-Bold	Base Font.
			Helvetica-Oblique	Base Font.
			Helvetica-BoldOblique	Base Font.
			Symbol	Type 1 font.
			Times-Roman	Type 1 font.
			Times-Bold	Base Font.
			Time-Italic	Base Font.
			Times-BoldItalic	Base Font.

	HeiseiMin-W3	CJK Font. Japanese.
	HeiseiKakuGo- W5	CJK Font. Japanese.
	STSong-Light	CJK Font. Chinese Simplified.
	MHei-Medium	CJK Font. Chinese Traditional.
	MSung-Light	CJK Font. Chinese Traditional.
	HYGoThic- Medium	CJK Font. Korean.
	HYSMyeongJo- Medium	CJK Font. Korean.
size	integer	Default. 10.
style	normal	Default. normal.
	Bold	
	Italic	
	line-through	
	Underline	
color	Color	Default. black.
encoding	Cp1250	Latin 2 Eastern Europe
	Cp1251	Cyrillic
	Cp1252	Default. Latin 1
	Cp1253	Greek
	Cp1254	Turkish
	Cp1257	Windows Baltic

	Identity-H	Unicode Horizontal.
	Identity-V	Unicode Vertical.
	UniJIS-UCS2-H	Japanese. Unicode (UCS-2) encoding for the Adobe-Japan character collection. Horizontal.
	UniJIS-UCS2-V	Japanese. Unicode (UCS-2) encoding for the Adobe-Japan character collection. Vertical.
	UniJIS-UCS2-HW-H	Japanese. Same as UniJIS-UCS2-H, but replaces proportional Latin characters with half-width forms. Horizontal.
	UniJIS-UCS2-HW-V	Japanese. Same as UniJIS-UCS2-V, but replaces proportional Latin characters with half-width forms. Vertical.
	UniGB-UCS2-H	Chinese Simplified. Horizontal.
	UniGB-UCS2-V	Chinese Simplified. Vertical.
	UniKS-UCS2-H	Korean. Horizontal.
	UniKS-UCS2-V	Korean. Vertical.
file	Value	Path of .TTF file.
background	Color	
embedded	Boolean	Default. false

Example

```
<style name="normal" font="Helvetica" size="10" style="normal" />
```

```
<style name="white" font="Helvetica" size="10" style="normal" color="white
```

```
<style name="Greek" file="/QIBM/ProdData/OS400/Fonts/TTFonts/mtsansdw  
H" embedded="true"/>
```

Element - annotation

The annotation element is a top-level element that defines an annotation.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value	
annotation	name	string		
	author	string		
	subject	string		
	date	string	yyyy-mm-dd HH:MM:SS+HH:MM The default is the current datetime.	
	icon	help		Default.
		comment		
		note		
		key		
		insert		
	color	color		
	type	square		Default.
		circle		
	value	string		Annotation text.

If the annotation text is large and extends over several lines then child phrase elements can be used.

Phrase text can come from the value attribute or text nodes between the start and end phrase tag.

The options attribute can be used to insert spaces or new lines before and after the text value.

The options value is a comma-separated list of tokens.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
phrase	options	sb	Space before text value.
		sa	Space after text value.
		nb	New line before text value.
		na	New line after text value.
	value	string	Text value.

Example

```
<annotation name="doc-  
help" icon="help" author="author" color="annotation">  
  <phrase options="na">This is help one line</phrase>  
  <phrase options="na" value="This is help two"/>  
  <phrase options="na">This is help three</phrase>  
</annotation>
```

Element - grid

The grid element is used to create a portrait and landscape grid pages that can be used for designing process.

The vertical and horizontal lines are 10 points apart and every 100 points a heavier line is drawn.

During the development process use the document grid attribute to overlay a grid on each page being created.

Syntax:

Element Attributes Value Notes for Element/Attribute/Value

grid

Example

<grid/>

Element - add

The add element is used to add a template to the current document page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
add	x1	integer	Default. 0.
	y1	integer	Default. 0.
	template	string	Name of template.
	border	boolean	Default. Document border.

Example

```
<add template="logo" x1="300" y1="400"/>
```

Element - import

The import element is used to import pages from another PDF document into the current document.

Use the <page> element to move off the last imported page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
import	file	value	Path of PDF file.
	page	String	Selected pages. A comma separated list of pages or page ranges. 10 10,14-16,23 Default. blank means all pages.
	cache	boolean	Default. true.
	file-exist	boolean	Default. false.
	page-exist	boolean	Default. false.
	file-mandatory	boolean	Default. false.
	background	color	Default. Current page background color.
	transform	a,b,c,d,e,f	Apply custom affine transform.
	directory	value	Optional. Import all PDF files in specified directory. All pages in each PDF file are imported.

Example

```
<import file="standard-contract.pdf"/>
```

```
<import file="standard-collection.pdf" page="2"/>
```


Element - import-image

The import-image element is used to import an image into the current document.

The image is scaled and center aligned to fit the page.

Use the <page> element to move off the new image page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
import-image	file	value	Path of image file.
	file-exist	Boolean	Default. false.
	file-mandatory	Boolean	Default. false.
	margin	value	left, right, top, bottom Default. Document margin.
	page-size	A0 to A4 B0 to B5 letter	Default. Document page.
		value	width, height
	orientation	portrait landscape	Default. Document orientation.

Example

```
<import-image file="map.jpeg" margin="0,0,0,0"/>
```

Element - attachment

The attachment element is used to add a file attachment to the current document.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
attachment	file	value	Path of attachment file.
	file-display	value	Attachment name in document. Default. Name of attachment file.
	file-description	value	
	file-exist	boolean	Default. false.
	file-mandatory	boolean	Default. false.
	file-date	value	File modification date. YYYY-MM-DD HH:MM:SS Default: Current datetime.
	file-compression	boolean	Default: true.

Example

```
<attachment file="CORD443.jpg" file-display="drill.jpg" file-  
description="Power Drill"/>
```

Element - page

The page element is used to create a new page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value	
page	page-size	A0 to A4	Default. Document page.	
		B0 to B5		
		letter		
		value		width, height
		margin		value
orientation		portrait	Default. Document orientation.	
		landscape		
background	color		Default. Document background.	

Example

```
<page/>
```

```
<page page-size="720,720" margin="40" orientation="landscape"/>
```

```
<page page-size="A4" margin="40,40,40,40" orientation="landscape"/>
```

Element - annotation

The annotation element is used to add defined annotation text to the current page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
annotation	x1	integer	Default. 0.
	y1	integer	Default. 0.
	name	string	Name of annotation.

Example

```
<annotation name="doc-help" x1="50" y1="550"/>
```

Element - anchor

The anchor element is used to a clickable region to the current page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
anchor	x1	integer	Default. 0.
	y1	integer	Default. 0.
	width	integer	Default. 0.
	height	integer	Default. 0.
	url	String	
	border	boolean	Default. Document border.

Example

```
<anchor x1="70" y1="540" width="50" height="30" url="http://www.lansa.co
```

Element - text-align

The text-align element is used to add aligned text to the current page.

Text align works by using the x1, y1 as a point around which the text aligns.

By default, x1 is the page width divided by 2 and y1 is the page height divided by 2. This means that the default alignment point is the center of the page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value	
text-align	x1	integer	Default. Page width / 2.	
	y1	integer	Default. Page height / 2.	
	value	string	Text	
	align	left		Default.
		center		
		right		
	rotation	integer	Default. 0.	
style	style	Default. Helvetica, 10, normal.		

Example

```
<text-align y1="400" align="center" value="heading"/>
```

```
<text-align style="gray" value="Blank Page" align="center" rotation="90"/>
```

Element - text

The text element is used to add text to the current page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value	
text	x1	integer	Default. 0.	
	y1	integer	Default. 0.	
	width	integer	Default. 0.	
	height	integer	Default. 0.	
	margin	integer	Default. 0.	
	border	Boolean	Default. Document border.	
	align	left		Default.
		center		
		right		
	style	style	Default. Helvetica, 10, normal.	
	leading	float	Default. fontsize * 1.5	
	value	string	Text.	
	anchor	url		
	annotation	string	Name of annotation	
	date-format	string	5.1.4 Date and Time Formats	
	decimal-format	string	5.1.5 Decimal Formats	
	locale	string	en_US fr_FR Default. default locale.	
use-list	Boolean	Default. false.		

	use-list-option	sb	Space before text value. Default. sb.
		sa	Space after text value.
		nb	New line before text value.
		na	New line after text value.

Text created from child phrase elements

If the text is large and extends over several lines then child phrase elements can be used.

Text created from child phrase elements take precedence over the text element value attribute.

Phrase text can come from the phrase element value attribute or a text node between the start and end phrase element tags.

The phrase element options attribute can be used to insert spaces or new lines before and after the text value. The options value is a comma-separated list of tokens.

Text created from list argument

The use-list attribute allows text to be created from a list.

This option takes precedence over the other methods of creating text.

The use-list-option attribute specifies the default text option value to be used.

The first list field contains the text and an optional second field is used to control the concatenation process.

If no second field is present or the field has a value of blank then the default text option value is used.

The possible option values are:

SB - Add a space before appending the text entry, except for the first entry.

NB - Add a new line before appending the text entry, except for the first entry.

SA - Add a space after appending the text entry, except for the last entry.

NA - Add a new line after appending the text entry.

If the option value is not one of the above then no additional action is done and the next entry is appended to the previous entries.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
phrase	options	sb	Space before text value.
		sa	Space after text value.
		nb	New line before text value.
		na	New line after text value.
value		string	Text value.
style		style	Default. Text element style.
anchor		url	
annotation		string	Name of annotation.
date-format		string	5.1.4 Date and Time Formats
decimal-format		string	5.1.5 Decimal Formats
locale		string	en_US fr_FR Default. default locale.

Example

```
<text x1="20" y1="700" width="100" height="20" value="some text"/>
```

```
<text x1="70" y1="500" width="200" height="400" style="text-modern">
```

```
  <phrase style="modern" options="na">First text phrase</phrase>
```

```
  <phrase style="modern-strike" options="na">Second text phrase</phrase>
```

```
  <phrase style="normal-
```

```
white" options="na" value="Third text phrase as a value attribute"/>
```

```
  <phrase options="na" anchor="http://www.lansa.com">Anchor</phrase>
```

```
  <phrase>Here is an</phrase>
```

```
  <phrase options="sb,sa" annotation="explain">annotation</phrase>
```

```
<phrase options="na">example</phrase>
<phrase options="sb,sb,sb,na,sa" value="{@date}" date-format="dd-MM-
yyyy"/>
<phrase options="na" value="{@date}" date-
format="dd/MM/yyyy hh:mm:ss"/>
<phrase options="na" value="23" decimal-format="0000"/>
<phrase options="na" value="34.56" decimal-format="#.###.##"/>
<phrase options="na" value="2334.56" decimal-
format="&#xA4;#.###.##"/>
</text>
```

Element - input field

The input element of type field is used to add an input field to the current page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
input	type	field	
	name	string	
	value	string	
	x1	integer	Default. 0.
	y1	integer	Default. 0.
	width	integer	Default. 0.
	height	integer	Default. 20.
	length	integer	Default. 0. A value of 0 indicates no input limit.
	style	style	Default. Helvetica, 10, normal.
	multiline	boolean	Default. false.
	readonly	boolean	Default. false.
	required	boolean	Default. false.
	export	boolean	Default. true.

Example

```
<input type="field" name="FIELD1" required="true" value="text" x1="100" y
```

```
<input type="field" name="FIELD2" multiline="true" readonly="true" export=
```

Element - input hidden

The input element of type hidden is used to add a hidden field to the current page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
input	type	hidden	
	name	string	
	value	string	
	length	integer	Default. 0. A value of 0 indicates no input limit.
	export	boolean	Default. true.

Example

```
<input type="hidden" name="FIELD3" value="SECRET" length="10"/>
```

Element - input password

The input element of type password is used to add an input password to the current page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
input	type	password	
	name	string	
	value	string	
	x1	integer	Default. 0.
	y1	integer	Default. 0.
	width	integer	Default. 0.
	height	integer	Default. 20.
	length	integer	Default. 0. A value of 0 indicates no input limit.
	style	style	Default. Helvetica, 10, normal.
	readonly	boolean	Default. false.
	required	boolean	Default. false.
	export	boolean	Default. true.

Example

```
<input type="password" name="FIELD4" x1="100" y1="660" width="30" len
```

Element - input checkbox

The input element of type checkbox is used to add an input checkbox to the current page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
input	type	checkbox	
	name	string	
	value	string	
	x1	integer	Default. 0.
	y1	integer	Default. 0.
	width	integer	Default. 12.
	height	integer	Default. 12.
	length	integer	Default. 0. A value of 0 indicates no input limit.
	selected	boolean	Default. false.
	readonly	boolean	Default. false.

Example

```
<input type="checkbox" name="CHECK1" value="CHK" selected="true" x1=
```

Element - input radio-group

The input element of type radio-group is used to logically group radio buttons.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
input	type	radio-group	
	name	string	
	length	integer	Default. 0. A value of 0 indicates no input limit.
	readonly	boolean	Default. false.

Example

```
<input type="radio-group" name="RADIO1" length="10">  
  <input type="radio" value="RAD1" x1="120" y1="540"/>  
  <input type="radio" value="RAD2" x1="140" y1="540" selected="true"/>  
  <input type="radio" value="RAD3" x1="160" y1="540"/>  
</input>
```

Element - input radio

The input element of type radio is used add a radio button to the current page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
input	type	radio	
	x1	integer	Default. 0.
	y1	integer	Default. 0.
	width	integer	Default. 12.
	height	integer	Default. 12.
	value	string	

Example

```
<input type="radio-group" name="RADIO1" length="10">  
  <input type="radio" value="RAD1" x1="120" y1="540"/>  
  <input type="radio" value="RAD2" x1="140" y1="540" selected="true"/>  
  <input type="radio" value="RAD3" x1="160" y1="540"/>  
</input>
```

Element - input combobox

The input element of type combobox is used add a combobox to the current page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
input	type	combobox	
	x1	integer	Default. 0.
	y1	integer	Default. 0.
	width	integer	Default. 0.
	height	integer	Default. 20.
	length	integer	Default. 0. A value of 0 indicates no input limit.
	style	style	Default. Helvetica, 10, normal
	readonly	boolean	Default. false.

Child item elements are used to add entries to the combobox.

The entry value can come from the item value attribute or as a text node between the start and end item tag.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
item	value	string	
	selected	boolean	Default. false.

Example

```
<input type="combobox" name="COMBO1" x1="100" y1="600" width="200"  
  <item value="C1"/>  
  <item>C2</item>  
  <item value="C3" selected="true"/>  
  <item value="C4"/>  
  <item value="{FIELD}"/>  
  <item value="C6"/>  
</input>
```

Element - input submit

The input element of type submit is used add a submit button to the current page.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
input	type	submit	
	name	string	Default. SUBMIT.
	x1	integer	
	y1	integer	
	width	integer	
	height	integer	Default. 20.
	length	integer	Default. 0. A value of 0 indicates no input limit.
	style	style	Default. Helvetica, 10, normal
	readonly	boolean	Default. false.
	caption	string	
	url	url	

Example

```
<input type="submit" caption="Submit" url="http://lansa01:1099/cgi-bin/jsmdirect?namevalue" x1="100" y1="510" width="50" style="form"/>
```

Element - table

The table element is used to add a table to the current page.

The working list argument is used to create the rows of table data.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
table	x1	integer	Default. 0.
	y1	integer	Default. 0.
	width	integer	Default. 0.
	height	integer	Default. 0.
	minimum-height	integer	Default. 0.
	fill	boolean	Default. false. Conditional. Requires height attribute. Use minimum-height to control height of empty rows.
	style	style	Default. Helvetica, 10, normal
	alternate	boolean	Default. false.
	alternate-color	color	
	title-show	boolean	Default. true.
	title-border-color	color	
	title-minimum-height	integer	Default. 0.
	border	boolean	Default. true.
	border-color	color	
	cell-border	boolean	Default. true.
	cell-border-	color	

color

Example

```
<table x1="20" y1="500" width="400" height="300" style="normal" alternate-  
  <column field="FIELD1" width-percentage="20" title="First Name"/>  
  <column field="FIELD2" width-  
percentage="20" title="Surname&#xA;Family Name"/>  
</table>
```

Element - column

The child column element is used to describe the table columns.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
column	title	string	
	title-style	style	
	title-vertical-align	top middle bottom	Default.
	title-horizontal-align	left center right	Default.
	title-border	boolean	Default. true.
	column-type	text image barcode	Default.
	style	style	Default. Table element style.
	width-percentage	integer	Default. 10.
	vertical-align	top middle bottom	Default.
	horizontal-align	left center right	Default.

border	boolean	Default. true.
alternate	boolean	Default. true.
alt-text	string	
alt-text-field	field	Name of list field.
type	barcode	Default. CODE128.
start	string	Default. A.
stop	string	Default. B.
start-stop	boolean	Default. true.
scale	integer	Default. 110.
rotation	integer	Default. 0.
field	field	Name of list field.
date-format	string	Date and Time Formats.
decimal-format	string	Decimal Formats.
substitute-newline	boolean	Substitute \n with a newline character. Default. false.
padding	string	Cell padding. Comma separated left,right,top,bottom values or a single value for all sides.
value	string	

Example

```
<table x1="20" y1="500" width="400" height="300" style="normal" alternate-
  <column field="FIELD1" width-percentage="20" title="First Name"/>
  <column field="FIELD2" width-
percentage="20" title="Surname&#xA;Family Name"/>
</table>
```


Element - list

The list element is used to add a list to the current page.

The list entries can come from a working list or if child item elements exist, then the item value attribute or child text nodes will be used. Child item elements take precedence over the working list.

If no list field is specified or the field does not exist in the list then the first field of the list is used.

Also a list can be created with out using a working list. Just use item elements with static values or bind single field values. Any value that is empty will be excluded from the list.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value	
list	x1	integer	Default. 0.	
	y1	integer	Default. 0.	
	width	integer	Default. 0.	
	style	style	Default. Helvetica, 10, normal	
	border	boolean	Default. Document border.	
	vertical-align	top		
		middle		Default.
		bottom		
	horizontal-align	left		Default.
		center		
		right		
	symbol	string		Default. bullet.
	symbol-default	boolean		Default. true.
field	field		Name of list field.	

Example

```
<list x1="20" y1="200" width="200" field="FIELD1" horizontal-align="right"/>
```

```
<list x1="20" y1="100" width="200">  
  <item value="Item1"/>  
  <item value="{FIELD2}"/>  
  <item style="modern">Item 2</item>  
  <item>Item 3</item>  
</list>
```

Element - item

The child item element is used to add list entries.

The entry value can come from the item value attribute or as a text node between the start and end item tag.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
item	value	string	
	style	style	Default. List element style.

Example

```
<list x1="20" y1="200" width="200" field="FIELD1" horizontal-align="right"/>
```

```
<list x1="20" y1="100" width="200">  
  <item value="Item1"/>  
  <item value="{FIELD2}"/>  
  <item style="modern">Item 2</item>  
  <item>Item 3</item>  
</list>
```

Element - if

The if element is used to add conditional logic.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
if	compare	string	
	value	string	
	operation	equal not_equal	Default.
	goto	content	Go to named content section.

Example

```
<if compare="{@page}" value="11" operation="equal" goto="additional"/>
```

```
<if compare="{@page}" value="12" operation="equal">
```

```
  <page/>
```

```
  <text x1="70" y1="300" width="100" height="20" value="some text"/>
```

```
</if>
```

Element - return

The return element ends the processing of the current content section.

Syntax:

Element Attributes Value Notes for Element/Attribute/Value

return

Example

```
<if compare="{@page}" value="12" operation="equal">  
  <return/>  
</if>
```

Element - line

The line element draws a line.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
line	x1	integer	Default. 0.
	y1	integer	Default. 0.
	x2	integer	Default. 0.
	y2	integer	Default. 0.
	line-width	integer	Default. 0.
	stroke-color	color	Default. black.

Example

```
<line x1="10" y1="290" x2="190" y2="290" stroke-color="blue"/>
```

Element - circle

The circle element draws a circle.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
circle	x1	integer	Default. 0.
	y1	integer	Default. 0.
	radius	integer	Default. 0.
	line-width	integer	Default. 0.
	stroke-color	color	Default. black.
	fill-color	color	

Example

```
<circle x1="100" y1="100" radius="50"/>
```

Element - rectangle

The rectangle element draws a rectangle.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
rectangle	x1	integer	Default. 0.
	y1	integer	Default. 0.
	width	integer	Default. 0.
	height	integer	Default. 0.
	radius	integer	Default. 0.
	line-width	integer	Default. 0.
	stroke-color	color	Default. black.
	fill-color	color	

Example

```
<rectangle x1="10" y1="100" width="180" height="40"/>
```

Element - image

The image element adds an image to the current page.

Width and height take precedence over scale. Width and height must be greater than 0 to take effect.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
image	x1	integer	
	y1	integer	
	width	integer	Default. 0.
	height	integer	Default. 0.
	file	file	Path of image file.
	scale	integer	Default. 80.
	rotation	integer	Default. 0.
	file-exist	boolean	Default. false.
	file-mandatory	boolean	Default. false.
	border	boolean	Default. Document border.
	page	integer	Default 0. TIFF image page. Use -1 to add all images.
	anchor	url	

Example

```
<image x1="70" y1="500" file="house.jpg" width="200" height="100"/>
```

Element - barcode

The barcode element creates a barcode.

Width and height take precedence over scale. Width and height must be greater than 0 to take effect.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value	
barcode	x1	integer	Default. 0.	
	y1	integer	Default. 0.	
	width	integer	Default. 0.	
	height	integer	Default. 0.	
	type		CODE39	
			CODE39EXT	
			CODE128	Default.
			CODE128UCC	
			EAN8	
			EAN13	
	CODABAR			
	POSTNET			
	PLANET			
	INTER25			
	QRCODE	Java JRE 6.0 or higher is required. Requires non zero width and height attribute.		
	value	string	Barcode value.	
	supplement	string	Supplement value. Used by EAN13.	

scale	integer	Default. 110. Not used by QRCODE.
rotation	integer	Default. 0.
align.	left	
	center	Default.
	right	
start	string	Default. A. Used by CODABAR.
stop	string	Default. B. Used by CODABAR.
start-stop	boolean	Default. true. Show start/stop text. Used by CODE39, CODE39EXT and CODABAR.
alt-text	string	Barcode alternative text.

Example

```

<barcode x1="20" y1="700" type="CODE39" value="12345ABCD"/>
<barcode x1="20" y1="600" type="CODE39EXT" value="12345" alt-
text="some text" />
<barcode x1="20" y1="500" type="CODE128" value="12345aBCD"/>
<barcode x1="20" y1="400" type="CODE128UCC" value="12345aBCD"/>
<barcode x1="20" y1="300" type="EAN8" value="1234567"/>
<barcode x1="20" y1="200" type="EAN13" value="641718311430"/>
<barcode x1="20" y1="100" width="200" height="50" type="EAN13" value='
<barcode x1="20" y1="700" type="CODABAR" value="4015638721939"/>
<barcode x1="20" y1="600" type="POSTNET" value="123456789"/>
<barcode x1="20" y1="500" type="PLANET" value="123456789"/>
<barcode x1="20" y1="400" type="INTER25" value="41-1200076041-
001"/>
<barcode x1="20" y1="300" type="INTER25" value="411200076041001"/>

```

<barcode x1="20" y1="600" type="QRCODE" width="100" height="100" val

Element - sign-box

The sign-box element adds a signature box to the current page. With the border enabled the height of the box is 30 points.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
sign-box	x1	integer	Default. 0.
	y1	integer	Default. 0.
	width	integer	Default. 0.
	message	string	
	line-width	integer	Default. 0.
	border	boolean	Default. Document border.

Example

```
<sign-box x1="90" y1="500" width="400" message="Sign in blue ink"/>
```

Element - date-box

The date-box element adds a date box to the current page. With the border enabled the height of the box is 30 points and the width is 120 points.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
date-box	x1	integer	Default. 0.
	y1	integer	Default. 0.
	day	string	
	month	string	
	year	string	
	line-width	integer	Default. 0.
	border	boolean	Default. Document border.

Example

```
<date-box x1="90" y1="300" day="dd" month="mm" year="yyyy"/>
```

Element - group-box

The group-box element adds a group box to the current page.

Use a group box to enclose input radio buttons.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value	
group-box	x1	integer	Default. 0.	
	y1	integer	Default. 0.	
	width	integer	Default. 0.	
	height	integer	Default. 0.	
	line-width	integer	Default. 0.	
	border	boolean	Default. Document border.	
	caption	string		
	caption-indent	integer	Default. 0.	
	caption-width	integer	Default. 0.	
	caption-height	integer	Default. 0.	
	align	left		
		center	Default.	
		right		
style	style	Default. Helvetica, 10, normal		
leading	float	Default. fontsize * 1.5.		

Example

```
<group-box x1="100" y1="560" width="200" height="40" caption="Types" caption-width="50" caption-height="20" caption-indent="10" style="form"/>
```


Element - signature

The signature element is used to add a signature to the PDF document. If the width, height or page is zero, then no visible signature rectangle is created.

Syntax:

Element	Attributes	Value	Notes for Element/Attribute/Value
signature	name	string	
	keystore	string	JKS keystore.
	password	string	JKS keystore password.
	alias	string	JKS keystore alias.
	page	integer	Default. 1.
	x1	integer	Default. 0.
	y1	integer	Default. 0.
	width	integer	Default. 0.
	height	integer	Default. 0.
	reason	string	
	location	string	
	text	string	
	date	string	5.1.4 Date and Time Formats
	signed	string	Default. self self verisign wincer

Example

```
<signature name="verify" keystore="pdf-  
sign.jks" password="password" alias="pdf" page="2" x1="100" y1="100" wid  
text" date="19610324" signed="self"/>
```

5.34.12 Example XML Content

The following XML content example illustrates how the elements are used.

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  Sample PDF XML document
-->
<!-- Document permissions require password -->
<document page-size="A4"
  background="cyan"
  orientation="portrait"
  border="true"
  author="Acme Corporation"
  title="Document Title"
  subject="Document Subject"
  keywords="word1, word2"
  creator=""
  hide-toolbar=""
  hide-menubar=""
  hide-windowui=""
  allow-printing="true"
  allow-copy="false"
  allow-modify-contents="false"
  allow-modify-annotations="false"
  allow-fillin="false"
  allow-screenreader="false"
  allow-assembly="false"
  allow-degraded-printing="false">
<content name="top">
  <!--
    Barcode examples
  -->
  <barcode x1="20" y1="700" type="CODE39" value="12345ABCD"/>
  <barcode x1="20" y1="600" type="CODE39EXT" value="12345"/>
  <barcode x1="20" y1="500" type="CODE128" value="12345aBCD"/>
  <barcode x1="20" y1="400" type="CODE128UCC" value="12345aBCD"/>
  <barcode x1="20" y1="300" type="EAN8" value="1234567"/>
  <barcode x1="20" y1="200" type="EAN13" value="641718311430"/>
```

```

<barcode x1="20" y1="100" type="EAN13" value="641718311430" supplier
<!--
    Design grid
-->
<grid/>
<!--
    Barcode examples
-->
<barcode x1="20" y1="700" type="CODABAR" value="4015638721939"/>
<barcode x1="20" y1="600" type="POSTNET" value="123456789"/>
<barcode x1="20" y1="500" type="PLANET" value="123456789"/>
<barcode x1="20" y1="400" type="INTER25" value="41-1200076041-
001"/>
<barcode x1="20" y1="300" type="INTER25" value="411200076041001"/>
<!--
    Import
    DO NOT do a new page before an import.
    The import internally does a new page before each page read.
    DO a new page after the import to move off the last imported page.
-->
<import file="other.pdf"/>
<page/>
<input type="field" name="FIELD1" required="true" readonly="" export=""
<input type="field" name="FIELD2" multiline="true" readonly="true" expo
<input type="hidden" name="FIELD3" value="SECRET" length="10"/>
<input type="password" name="FIELD4" x1="100" y1="660" width="30" l
<input type="checkbox" name="CHECK1" readonly="" value="CHK" selec
<input type="combobox" name="COMBO1" readonly="" x1="100" y1="60
  <item value="C1"/>
  <item>C2</item>
  <item value="C3" selected="true"/>
  <item value="C4"/>
  <item value="{XYZ}"/>
  <item value="C6"/>
</input>
<input type="radio-group" name="RADIO" readonly="" length="10">
  <input type="radio" value="RAD1" x1="120" y1="540"/>
  <input type="radio" value="RAD2" x1="140" y1="540" selected="true"/>
  <input type="radio" value="RAD3" x1="160" y1="540"/>

```

```

</input>
<group-
box x1="100" y1="560" width="200" height="40" caption="Types" caption-
width="50" caption-height="20" caption-indent="10" style="form"/>
  <input type="submit" caption="Submit" url="http://lansa01:88/cgi-
bin/jsmdirect?namevalue" x1="100" y1="510" width="50" style="form"/>
  <page/>
</content>
<content name="table">
  <!--
    Table using working list
  -->
  <text x1="20" y1="700" width="100" height="20" style="normal" border=""
{@rowcount}"/>
  <annotation name="doc-help" x1="50" y1="550"/>
  <table x1="20" y1="500" width="400" style="normal" alternate="true" alter-
color="" title-show="">
    <column field="FNAME" width-percentage="20" style="italic" vertical-
align="top" horizontal-align="right" title-horizontal-align="left" title-
border="" title-style="title-beach" title="First Name"/>
    <column field="SNAME" width-percentage="20" vertical-
align="bottom" horizontal-align="left" title-horizontal-align="right" title-
border="" title="Surname&#xA;Family Name"/>
    <column column-type="image" field="IMAGE" alt-text="none" alt-text-
field="FNAME" width-
percentage="20" title="Image" value="adobe.gif" scale="70"/>
    <column column-type="barcode" field="ID" width-
percentage="20" type="CODE128" scale="" rotation="" start="" stop="" start-
stop="" alternate="false" border="false" horizontal-align="center" title-
horizontal-align="" title-border="false" title=""/>
  </table>
  <!--
    List using working list data
  -->
  <list x1="20" y1="200" width="200" style="modern" field="SNAME" borde-
align="right"/>
  <!--
    List using items
  -->

```

```

<list x1="20" y1="100" width="200" style="italic" border="" horizontal-
align="">
  <item value="Item1"/>
  <item value="{XYZ}"/>
  <item style="modern">Item 2</item>
  <item>Item 3</item>
</list>
<!--
  New page
-->
<page/>
</content>
<content name="detail">
  <!--
    Text area
  -->
  <text x1="70" y1="500" width="200" height="400" style="text-
modern" border="" align="left" leading="">
    <phrase style="modern" options="na">This is the first text phrase</phrase>
    <phrase style="modern-
strike" options="na">This is the second text phrase</phrase>
    <phrase style="normal-
white" options="na" value="This is the third text phrase as a value attribute"/>
    <phrase options="na" anchor="http://www.lansa.com">anchor</phrase>
    <phrase>Here is an</phrase>
    <phrase options="sb,sa" annotation="explain">annotation</phrase>
    <phrase options="na">example</phrase>
    <phrase options="sb,sb,sb,na,sa" value="{@date}" date-format="dd-MM-
yyyy"/>
    <phrase options="na" value="123.45" decimal-
format="&#xA4;#,###.##"/>
  </text>
  <!--
    New page
  -->
  <page/>
  <!--
    Add template
  -->

```

```

<add template="logo" x1="20" y1="800"/>
<add template="logo" x1="20" y1="400"/>

<add template="image" x1="300" y1="800"/>
<add template="logo" x1="300" y1="400"/>
<!--
    New page
-->
<page/>
<text x1="70" y1="600" width="40" height="16" style="normal-
white" border="" leading="12" align="center" value="1234"/>
<text x1="70" y1="550" width="100" height="16" style="normal-
white" border="" leading="12" align="center" value="Go to LANSA" anchor=
<!--
    Signature box and date box
-->
<sign-box x1="90" y1="500" width="400" line-
width="" message="Sign in blue ink" border=""/>
<date-box x1="90" y1="300" line-
width="" day="dd" month="mm" year="yyyy" border=""/>

<page/>
<!--
    Image
-->
<image x1="70" y1="500" file="univac.jpg" scale="80"/>

</content>
<content name="bottom">
<page orientation="landscape"/>
<text-align x1="100" y1="500" style="italic-
blue" align="left" value="Some aligned text 1" rotation="0"/>
<text x1="70" y1="300" width="100" height="20" border="" align="left" lea
{@page}"/>
<!--
    Condition logic
-->
<if compare="{@page}" value="11" operation="equal" goto="additional"/>
<if compare="{@page}" value="12" operation="equal">

```

```

<if compare="3" value="12" operation="not_equal">

  <page background="yellow"/>
  <text-
align x1="100" y1="500" style="beach" align="left" value="Some aligned text"
  <return/>
  </if>
</if>
  <text x1="70" y1="300" width="100" height="20" border="" align="left" lea
{@page}"/>
</content>
<content name="additional">
  <page/>
  <text x1="70" y1="400" width="400" height="40" style="beach" border=""
  <text x1="70" y1="300" width="400" height="40" style="smokin" border=""
  <text x1="70" y1="200" width="400" height="40" style="smokin-
condensed" border="" align="left" leading="" value="additional"/>
</content>
<content name="graphic">
  <!--
    Graphic line, rectangle, circle
    These co-ordinates are based on the template rectangle
    <circle x1="100" y1="100" radius="50" stroke-color="" fill-
color="" />
    -->
    <line x1="10" y1="290" x2="190" y2="290" stroke-color="blue" line-
width=""/>
    <line x1="10" y1="10" x2="190" y2="10" stroke-color="red" line-
width=""/>
  </content>
  <!--
    Templates
    co-ordinates are based on the template rectangle
    -->
  <template name="banner" width="200" height="300"/>
  <template name="logo" width="200" height="300">
    <rectangle x1="10" y1="100" width="180" height="40" radius="" fill-
color="" stroke-color="" line-width=""/>
  </template>

```

```

<template name="image" width="200" height="300">
  <grid/>
  <page/>
  <import />
  <image x1="0" y1="300" file="univac.jpg" scale="80"/>
</template>
<!--
  Custom colors
-->
<color name="charcoal" rgb="100,80,90"/>
<color name="annotation" rgb="251,248,104"/>
<!--
  Custom styles
-->
<style name="default" font="Courier" size="10" style="bold" color="charcoal" />
<style name="form" font="Times-Roman" size="10"/>
<style name="normal" font="Helvetica" size="10" style="normal" color="black" />
<style name="normal-white" font="Helvetica" size="10" style="normal" color="white" background="beach" file="miami.ttf" size="15" color="black" background="yellow" encoding="Cp1252" />
<style name="title-beach" file="miami.ttf" size="15" color="black" encoding="Cp1252" />
<style name="smokin" file="marlbo.ttf" size="15" color="blue" encoding="Cp1252" />
<style name="smokin-condensed" file="marlboc.ttf" size="15" color="black" encoding="Cp1252" />
<style name="italic" font="Helvetica" size="10" style="italic" color="black" />
<style name="strike" font="Helvetica" size="10" style="line-through" color="black"/>
<style name="text-modern" font="Times-Roman" size="12" style="normal" color="black" background="yellow"/>
<style name="modern" font="Times-Roman" size="12" style="bold italic" color="red"/>
<style name="modern-strike" font="Times-Roman" size="12" style="line-through" color="black"/>
<style name="italic-blue" font="Times-Italic" color="blue" size="20" encoding="Cp1252"/>
<style name="beach" file="miami.ttf" size="30" color="black" encoding="Cp1252" />
<!--
  Annotations

```

```
-->
<annotation name="doc-
help" icon="help" author="author" color="annotation">
  <phrase options="na">This is help one line</phrase>
  <phrase options="na" value="This is help two"/>
  <phrase options="na">This is help three</phrase>
</annotation>
<annotation name="explain" type="circle" color="red">
  <phrase options="na">This explains this annotation</phrase>
  <phrase options="na" value="More explaining"/>
</annotation>
</document>
```

5.34.13 Example RDML function

The following RDML function example illustrates how to use the example XML content.

```
***** Beginning of RDML commands *****
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
*****

DEFINE FIELD(#ID) TYPE(*CHAR) LENGTH(5)
DEFINE FIELD(#FNAME) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#SNAME) TYPE(*CHAR) LENGTH(30)
*****

DEF_LIST NAME(#TBLLST) FIELDS((#ID) (#FNAME) (#SNAME)) TYP
*****

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SE
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****

CHANGE FIELD(#JSMCMD) TO(''CREATE DOCUMENT(report.pdf) CC
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****

USE BUILTIN(JSM_COMMAND) WITH_ARGS('ADD CONTENT(TO
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****

BEGIN_LOOP TO(2)
*****

CLR_LIST NAMED(#TBLLST)
*****

CHANGE FIELD(#ID) TO(A1001)
CHANGE FIELD(#FNAME) TO(''Alick'')
CHANGE FIELD(#SNAME) TO(''Buckley'')
ADD_ENTRY TO_LIST(#TBLLST)
*****

CHANGE FIELD(#ID) TO(A1002)
```

```

CHANGE  FIELD(#FNAME) TO("John")
CHANGE  FIELD(#SNAME) TO("Smith")
ADD_ENTRY TO_LIST(#TBLLST)
*****

CHANGE  FIELD(#ID) TO(A1003)
CHANGE  FIELD(#FNAME) TO("Jack")
CHANGE  FIELD(#SNAME) TO("Brown")
ADD_ENTRY TO_LIST(#TBLLST)
*****

CHANGE  FIELD(#ID) TO(A1004)
CHANGE  FIELD(#FNAME) TO("Billy")
CHANGE  FIELD(#SNAME) TO("Moylan")
ADD_ENTRY TO_LIST(#TBLLST)
CHANGE  FIELD(#JSMCMD) TO('ADD CONTENT(TABLE) SERVICE_I
USE     BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****

END_LOOP
*****

USE     BUILTIN(JSM_COMMAND) WITH_ARGS('ADD CONTENT(GR
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****

USE     BUILTIN(JSM_COMMAND) WITH_ARGS('ADD CONTENT(DE
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****

USE     BUILTIN(JSM_COMMAND) WITH_ARGS('ADD CONTENT(BO
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****

USE     BUILTIN(JSM_COMMAND) WITH_ARGS('CLOSE') TO_GET(#J
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****

USE     BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****

***** SUB ROUTINES
*****

SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSMMSG
*****

IF     COND('#JSMSTS *NE OK')

```

```
DISPLAY  FIELDS((#JSMSTS) (#JSMMSG))
*****
USE      BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****
MENU     MSGTXT('Java service error has occurred')
*****
*****
ENDIF
*****
ENDROUTINE
***** End of RDML commands *****
```

5.35 SVFileService

The SVFileService provides support for an application to create or read separated value files.

Separated variable files are widely used to represent and exchange "flat" database information. Comma-separated values (CSV) and Tab-separated variables (TSV) are automatically supported or you can specify a value to be interpreted as the separator.

Related Services

The SVFileService is not dependent on other services.

The POP3MailService overlaps the SVFileService to some extent. The READ command on the POP3MailService also allows processing of separator delimited files. Assuming the separated variable file you want to process is attached to an incoming email, the POP3MailService may be adequate for handling the processing of the file.

Similarly when SVFileService reads from or writes to a remote server an FTP connection is established to connect to the remote server. In this instance the SVFileService overlaps the FTPService to some extent.

Technical Specifications

When reading or writing from a remote file server, the SVFileService uses FTP to connect to the remote server.

5.35.1 What can I use the SVFileService for?

Using the SVFileService an application can directly write columnar information to a separated variable file, or read information from a separated variable file into a list for further processing.

By default the SVFileService processes files from the local file system but can be configured to use FTP to establish a connection to a remote file server for access to remote files.

When the database support is used, database level commitment control is fully supported. The service also provides a set of commands for performing housekeeping operations on the contents of the file system (such as renaming and deleting files).

5.35.2 Using the SVFileService

Whether you are writing your client application in LANSAs RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application that reads a separated variable file would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

READ

SERVICE_UNLOAD

JSM(X)_CLOSE

A similar sequence of commands would be used to write to a separated variable file with the READ command replaced by a WRITE command.

Refer to [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.35.3 SVFileService Commands

Your application issues commands to the SVFileService by passing the command strings through the Java Service Manager using the JSM_COMMAND or JSMX_COMMAND Built-In Function, or an equivalent Built-In Function or API for your chosen development language.

The commands that the SVFileService processes are:

SERVICE_LOAD

READ

WRITE

DELETE

RENAME

CONNECT

SET

LIST

COMMIT

ROLLBACK

DISCONNECT

SERVICE_GET

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
---->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

- | | |
|---------|--|
| SERVICE | The name of the service to be loaded - in this case SVFileService. |
| TRACE | To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.
The possible values for the TRACE keyword are: <ul style="list-style-type: none">• *NO• *YES• *ERROR The trace option *ERROR will turn on tracing and if the |

service does not return an ERROR or FATAL status to the client, the trace files are deleted on the JSM CLOSE call.

TRACE_NAME The optional TRACE_NAME keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(SVFILESEI  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jsmcommand := 'service_load service(SVFileService)  
use builtin(jsmx_command) with_args(#jsmxhdl1 #jsmcommand) to_get(#jsn
```

READ

The READ command processes the nominated FILE and returns the information in a working list (unless the SVTABLE keyword is used).

To read a file from the local file system only requires the FILE keyword.

To read a file from a remote file system using the FTP protocol requires the HOST keyword with appropriate USER, PASSWORD and access details for the remote server.

The service also determines the encoding to apply to the read byte content to convert it to Unicode content and determines if the Unicode data needs to be archived.

Required

READ ----- *FILE* ----- *file path* ----->

Conditional

>-- *HOST* ----- *host:port* ----->

>-- *USER* ----- *value* ----->

>-- *PASSWORD* ----- *value* ----->

>-- *NAMEFMT* ----- *none* ----->

0

1

>-- *DATALINK* ----- **PASV* ----->

**PORT*

>-- *MODE* ----- **BINARY* ----->

**ASCII*

Optional

>-- *CONTENT* ----- **CSV* ----->

```

    *TSV
    *SV

>-- SEPARATOR ----- value ----->
    *COMMA
    *SEMICOLON
    *TAB
    *TILDE

>-- ENCODING ----- *DEFAULT ----->
    value

>-- SVROW ----- value ----->

>-- SVROWLIMIT ----- value ----->
    *NONE
    *LIST
    *AVAILABLE

>-- SVMODE ----- *NONE ----->
    *IGNORE
    *USE

>-- SVHEAD ----- value ----->

>-- SVLABEL ----- value ----->

>-- SVTABLE ----- table name ----->

>-- SVCOLUMN ----- value ----->

>-- NUMBERFORMAT --- *NONE -----
->
    *DEFAULT
    *CLIENT
    *USERAGENT
    value

>-- ARCHIVE ----- file path -----|
```

Keywords

FILE	<p>Nominate the path and file name.</p> <p>This value can be a relative or absolute path. If the path is relative, the current working directory is the JSM instance directory.</p>
HOST	<p>This keyword is only required when reading from a remote file system.</p> <p>Nominate a FTP server to connect to. The FTP server can be specified as an IP address, nnn.nnn.nnn.nnn:port, or a domain name.</p> <p>If a port number is not supplied on an IP address the default value 21 is used unless the keyword SECURE (*IMPLICIT) is used in which case the default port value is 990.</p>
USER	<p>Only required when reading from a remote file system.</p>
PASSWORD	<p>Only required when reading from a remote file system.</p> <p>The password for the database server that corresponds to the USER keyword.</p>
NAMEFMT	<p>Only valid when reading from a remote file system.</p> <p>A value of 0 indicates an IBM i path name format library/file.member is to be used.</p> <p>A value of 1 indicates a Windows path name format /directory/directory/file is to be used.</p>
DATALINK	<p>Only valid when reading from a remote file system.</p> <p>The possible values are *PASV (Passive) or *PORT (Port). *PASV is the default value.</p>
MODE	<p>Only valid when reading from a remote file system.</p> <p>The default value is *BINARY. Use a value of *ASCII if required.</p>
CONTENT	<p>By default the nominated file is processed as *CSV.</p> <p>Alternately the content can be processed as a Tab</p>

separated variables (*TSV) or separated by a variable (*SV) as specified in the separator keyword. If processing content that includes a separator variable, refer to the [SEPARATOR](#), [SVHEAD](#), [SVMODE](#), [SVTABLE](#), [SVCOLUMN](#), [SVLABEL](#), [NUMBERFORMAT](#), [TRIM](#) and [TRUNCATE](#) keywords for additional processing options.

The working list used to store the content must be defined with an appropriate number of columns to store the data.

- SEPARATOR** This keyword is to indicate what character is used as a separator. The separator does not need to be defined if the content is indicated as *CSV or *TSV. If the SEPARATOR keyword is present and no CONTENT keyword is supplied CONTENT(*SV) is assumed. Refer to [SEPARATOR](#) for more information.
- ENCODING** ENCODING is used to specify what encoding must be applied to a byte content to convert it to a Unicode string. The default value for the ENCODING keyword is *DEFAULT. Refer to [ENCODING](#) for more information.
- SVROW** The SVROW keyword is used to specify the starting data record row. Refer to [SVROW](#) for more information,
- SVROWLIMIT** The SVROWLIMIT keyword is used to specify the number of data records to read. The possible values are *NONE, *LIST, *AVAILABLE or an integer value. Refer to [SVROWLIMIT](#) for more information.
- SVMODE** The SVMODE keyword is used by content handlers and services that process separated value data to indicate how to handle the inbound separated value data. Refer to [SVMODE](#) for more information.

SVHEAD	<p>The optional keyword SVHEAD is used to describe the field layout of the separated value data.</p> <p>Refer to SVHEAD for more information.</p>
SVTABLE	<p>If the SVTABLE keyword is present then the separated variable file data is inserted into the specified table using the current database connection. If no value is provided for SVTABLE the file data is returned in the working list argument.</p> <p>Refer to SVTABLE for more information.</p>
SVCOLUMN	<p>SVCOLUMN is only used in conjunction with the SVTABLE keyword. The SVCOLUMN separates value services to define the relational database columns. A look up on the service properties resource is done using the sv.column.{value}.</p> <p>Refer to SVCOLUMN for more information.</p>
SVLABEL	<p>SVLABEL is only used in conjunction with the SVTABLE keyword. SVLABEL is used to include the keyword value as the first column value for database table inserts or as the where constraint for database table selects.</p> <p>Refer to SVLABEL for more information.</p>
NUMBERFORMAT	<p>The optional keyword NUMBERFORMAT is used to handle numeric strings, where the decimal separator is not the decimal point character ".".</p> <p>Refer to NUMBERFORMAT for more information.</p>
ARCHIVE	<p>Use the optional keyword ARCHIVE to nominate the path and file name used to archive content.</p> <p>Refer to ARCHIVE for more information.</p>

Examples

RDML

```
DEF_LIST NAME(#WRKLIST) FIELDS(#LINENUM #PARTNUM #PARTL
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('READ FILE(order.csv) SE
```

or

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('READ FILE(order.csv) SE  
) TO_GET(#JSMSTS #JSMMSG #WRKLIST)
```

or

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('READ FILE(order.csv) SV
```

RDMLX

```
#jsmcmd := 'read file(' + #jsmfile + ') content(*sv) separator(' + #separator + ')  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

WRITE

The WRITE command selects information from a working list (or table if the SVTABLE keyword is used) and saves it to the separated variable file nominated by the FILE keyword.

To write a file to the local file system only requires the FILE keyword.

To write a file to a remote file system using the FTP protocol requires the HOST keyword with appropriate USER, PASSWORD and access details for the remote server.

The service determines the encoding to apply to the Unicode content to convert it to byte content and determines if the Unicode data needs to be archived.

Required

WRITE ----- *FILE* ----- *file path* ----->

Conditional

>-- *HOST* ----- *host:port* ----->

>-- *USER* ----- *value* ----->

>-- *PASSWORD* ----- *value* ----->

>-- *NAMEFMT* ----- *none* ----->

0

1

>-- *DATALINK* ----- **PASV* ----->

**PORT*

>-- *MODE* ----- **BINARY* ----->

**ASCII*

Optional

>-- *CONTENT* ----- **CSV* ----->

**TSV*

**SV*

>-- *SEPARATOR* ----- *value* ----->

**COMMA*

**SEMICOLON*

**TAB*

**TILDE*

>-- *ENCODING* ----- **DEFAULT* ----->

value

>-- *SVQUOTE* ----- **NONE* ----->

**TEXT*

**ALL*

>-- *SVHEAD* ----- *value* ----->

>-- *SVEXCLUDE* ----- *value* ----->

>-- *SVLABEL* ----- *value* ----->

>-- *SVTABLE* ----- *table name* ----->

>-- *SVCOLUMN* ----- *value* ----->

>-- *APPEND* ----- **YES* ----->

**NO*

>-- *NUMBERFORMAT* --- **NONE* -----

->

**DEFAULT*

**CLIENT*

**USERAGENT*

value

>-- *ARCHIVE* ----- *file path* -----|

Keywords

FILE	<p>Nominate the path and file name.</p> <p>This value can be a relative or absolute path. If the path is relative, the current working directory is the JSM instance directory.</p>
HOST	<p>The HOST keyword is only required when reading from a remote file system.</p> <p>Nominate a FTP server to connect to. The FTP server can be specified as an IP address, nnn.nnn.nnn.nnn:port, or a domain name.</p> <p>If a port number is not supplied on an IP address the default value 21 is used unless the keyword SECURE (*IMPLICIT) is used in which case the default port value is 990.</p>
USER	<p>The USER keyword is only required when reading from a remote file system.</p>
PASSWORD	<p>The PASSWORD keyword is only required when reading from a remote file system.</p> <p>The password for the database server that corresponds to the USER keyword.</p>
NAMEFMT	<p>The NAMEFMT keyword is only valid when reading from a remote file system.</p> <p>A value of 0 indicates an IBM i path name format library/file.member is to be used.</p> <p>A value of 1 indicates a Windows path name format /directory/directory/file is to be used.</p>
DATALINK	<p>The DATALINK keyword is only valid when reading from a remote file system.</p> <p>The possible values are *PASV (Passive) or *PORT (Port). *PASV is the default value.</p>
MODE	<p>The MODE keyword is only valid when reading from a remote file system.</p> <p>The default value is *BINARY. Use a value of *ASCII</p>

if required.

CONTENT

By default the nominated file is processed as *CSV.

Alternately the content can be processed as a Tab separated variables (*TSV) or separated by a variable (*SV) as specified in the separator keyword. If processing content that includes a separator variable refer to the [SEPARATOR](#), [SVHEAD](#), [SVMODE](#), [SVTABLE](#), [SVCOLUMN](#), [SVLABEL](#), [NUMBERFORMAT](#), [TRIM](#) and [TRUNCATE](#) keywords for additional processing options.

The working list used to store the content must be defined with an appropriate number of columns to store the data.

SEPARATOR

The keyword SEPARATOR is to indicate what character is used as a separator. The separator does not need to be defined if the content is indicated as *CSV or *TSV.

If the SEPARATOR keyword is present and no CONTENT keyword is supplied CONTENT(*SV) is assumed.

Refer to [SEPARATOR](#) for more information.

ENCODING

ENCODING is used to specify what encoding must be applied to a byte content to convert it to a Unicode string. The default value for the ENCODING keyword is *DEFAULT.

Refer to [ENCODING](#) for more information.

APPEND

APPEND is used to append content to an existing local file. The default value for the APPEND keyword is *NO.

SVQUOTE

The optional keyword SVQUOTE is used to explicitly double quote values. The default value for SVQUOTE is *NONE. A value of *ALL means that all values are double quoted. A value of *TEXT means only text values are double quoted.

SVHEAD	<p>The optional keyword SVHEAD is used to describe the field layout of the separated value data.</p> <p>Refer to SVHEAD for more information.</p>
SVEXCLUDE	<p>The optional keyword SVEXCLUDE is used to exclude fields from the working list data. The value is one or more comma-separated working list field names.</p> <p>Refer to SVEXCLUDE for more information.</p>
SVTABLE	<p>If the SVTABLE keyword is present the data is selected from the specified table using the current database connection. If no value is provided for SVTABLE the file data is retrieved from the working list argument.</p> <p>Refer to SVTABLE for more information.</p>
SVCOLUMN	<p>SVCOLUMN is only used in conjunction with the SVTABLE keyword. SVCOLUMN separates value services to define the relational database columns.</p> <p>A look up on the service properties resource is done using the sv.column.{value}.</p> <p>Refer to SVCOLUMN for more information.</p>
SVLABEL	<p>SVLABEL is only used in conjunction with the SVTABLE keyword. SVLABEL is used to include the keyword value as the first column value for database table inserts or as the where constraint for database table selects.</p> <p>Refer to SVLABEL for more information.</p>
NUMBERFORMAT	<p>The optional keyword NUMBERFORMAT is used to handle numeric strings, where the decimal separator is not the decimal point character ".".</p> <p>Refer to NUMBERFORMAT for more information.</p>
ARCHIVE	<p>Use the optional keyword ARCHIVE to nominate the path and file name used to archive content.</p> <p>Refer to ARCHIVE for more information.</p>

Examples

RDML

```
DEF_LIST NAME(#WRKLIST) FIELDS(#LINENUM #PARTNUM #PARTI
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('WRITE FILE(order.csv) S
```

or

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('WRITE FILE(order.csv) S
```

or

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('WRITE FILE(order.csv) S
```

RDMLX

```
def_list name(#valueslst) fields(#std_num #std_obj #std_desc #std_amnt #std_
```

```
#jsmcmd := 'write file(' + #jsmfile + ') content(*sv) separator(' + #separator + ')
```

```
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

DELETE

The DELETE command deletes the specified local file.

```
DELETE ----- FILE ----- file path -----|
```

Keywords

FILE Nominate the file to be deleted. This value can be a relative or absolute path.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('DELETE FILE(/upload/or
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'delete file(/upload/order.cs
```

RENAME

The RENAME command renames the specified local file to a new name.

```
RENAME ----- FROM ----- file path ----->
      >-- TO ----- file path ----->
      >-- REPLACE ----- *NO -----|
                *YES
```

Keywords

- FROM** Nominate the path and file name to be renamed. The path and file name must exist.
- TO** Nominate the new path and file name. The path name must exist. The TO file must be in the same directory as the original FROM file.
- REPLACE** The default value is *NO. To allow a file to be replaced by the renamed file this keyword must be set to *YES.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('RENAME FROM(order.xi
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'rename from(order.xml) to
```

CONNECT

The CONNECT command connects to a remote database. By default a connection object is in auto-commit mode, which means that it automatically commits changes after executing each statement.

```
CONNECT ----- DRIVER ----- value ----->
>-- DATABASE ----- value ----->
>-- USER ----- value ----->
>-- PASSWORD ----- value -----|
```

Keywords

DRIVER The symbolic name of the database driver used to locate the server. For example:

```
CONNECT DRIVER(SQLSERVER) DATABASE(NORTHW
```

Searches for a corresponding property like:

```
driver.sqlserver=com.ddtek.jdbc.sqlserver.SQLServerDriver
```

DATABASE The symbolic name of the database used to locate the service's property. For example:

```
CONNECT DRIVER(SQLSERVER) DATABASE(NORTHW
```

Searches for a corresponding property like:

```
database.northwind=jdbc:datadirect:sqlserver://99.99.99.99:12
```

USER The user profile to be used to log into the database server.

PASSWORD The password for the database server that corresponds to the US

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('CONNECT DRIVER(SQI
```

RDMLX

```
use builtin(jsmx_command) with_args(#jshmhandle 'connect driver(sqlserver) d
```

SET

The SET command is used for two distinct operations. Use the DIR keyword to set the current working directory or alternately use the READONLY, AUTOCOMMIT and ISOLATION keywords to define the level of access allowed to the current database connection.

Conditional

```
SET ----- DIR ----- directory path ----->

>-- READONLY ----- *NO ----->
      *YES

>-- AUTOCOMMIT ----- *YES ----->
      *NO

>-- ISOLATION ----- *NONE -----|
      *READCOMMITTED
      *READUNCOMMITTED
      *REPEATABLEREAD
      *SERIALIZABLE
```

Keywords

- | | |
|----------|--|
| DIR | Use the DIR keyword to nominate a relative or absolute directory path to be set as the current directory.
The DIR keyword does not relate to the other SET keywords. |
| READONLY | Indicate if the current database connection should allow read or write access. A database connection must be established using the CONNECT command. This keyword then applies to the current database connection.
By default when a database connection is established, read and write access are permitted (depending on restrictions imposed by the database).
A value of *YES indicates any changes to the database are |

automatically committed after executing each statement.

A value of *NO indicates that changes to the database are controlled by programmed transaction boundaries.

The READONLY keyword does not have any relationship with the DIR keyword.

AUTOCOMMIT Indicate if information should be automatically committed to the database. A database connection must be established using the CONNECT command to use this keyword. This keyword then applies to the current database connection. By default when a connection is established it is in auto-commit mode.

A value of *YES indicates any changes to the database are automatically committed after executing each statement.

A value of *NO indicates that changes to the database are controlled by programmed transaction boundaries.

The AUTOCOMMIT keyword does not have any relationship with the DIR keyword.

ISOLATION Indicate the level of transaction isolation to be applied to the current database. A database connection must be established using the CONNECT command to use this keyword. This keyword then applies to the current database connection.

The ISOLATION keyword does not have any relationship with the DIR keyword.

The default value is *NONE otherwise you can choose from these four levels of transaction defined by the ANSI/ISO SQL standard:

***READUNCOMMITTED**

All uncommitted data is readable from any connection. This is the same as not having any isolation (*NONE).

***READCOMMITTED**

This prevents dirty reads but does not prevent phantoms or non-repeatable reads. Using this isolation level, only data committed before the current transaction began will be available. Any dirty data or changes made by concurrent

transactions will not be available.

This level is obviously more restrictive than the *READUNCOMMITTED.

***REPEATABLEREAD**

This prevents dirty and non-repeatable reads but does not prevent phantom rows. This means the probability of other transactions having to wait for this one are increased when compared to *READUNCOMMITTED and *READCOMMITTED

This is more restrictive than *READCOMMITTED.

***SERIALIZABLE**

*SERIALIZABLE provides the highest transaction isolation. When a transaction is isolated at the *SERIALIZABLE level, only data committed before the transaction began is available. Neither dirty data nor concurrent transaction changes committed during transaction execution are available. This level emulates serial transaction execution, as transactions will effectively be executed one after another rather than concurrently.

This is more restrictive than *REPEATABLEREAD.

In relation to these isolation levels there are three phenomena that you need to understand before you can determine the correct isolation level to apply to your application, namely:

- Dirty Reads - A transaction reads data written by an uncommitted transaction. If the second transaction is rolled back, the data read by the first transaction is then invalid because the rollback undoes the changes. The first transaction won't be aware that the data it has read has become invalid.
- Non-repeatable Reads - A transaction re-reads data it has previously read and finds that data has been modified by another committed transaction.
- Phantom Read - Phantom reads occur when new records added to the database are detectable by transactions that started prior to the insert. A transaction re-executes a

query and returns a set of rows satisfying a search condition only to find that additional rows satisfying the condition have been inserted by another committed transaction.

The ANSI/ISO SQL standard isolation levels and the corresponding behaviors are summarized in the following table:

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read
*READUNCOMMITTED	Possible	Possible	Possible
*READCOMMITTED	Not possible	Possible	Possible
*REPEATABLEREAD	Not possible	Not possible	Possible
*SERIALIZABLE	Not possible	Not possible	Not possible

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET READONLY(*NO) I
```

RDMLX

```
use builtin(jsmx_command) with_args(#jshandle 'set dir(/newfiles') to_get(#j
```

LIST

The LIST command returns a working list of absolute file names in a directory. The working list to return the file names must be defined with a single field long enough to contain the absolute file name.

If no DIR keyword is provided and the current directory has not be changed using other commands, the default JSM instance directory will be searched. Sub-directories are not searched.

If no EXT keyword is provided all files in the directory will be searched by default.

```
LIST ----- DIR ----- directory path ----->
>-- EXT ----- file extension ----->
>-- SORT ----- *NONE ----->
      *NAME
      *MODIFIED
>-- REVERSE ----- *YES -----|
      *NO
```

Keywords

DIR Nominate a relative or absolute directory path to be searched. Sub-directories are not searched.

EXT Only select files with a nominated file extension. The filtering match is not case sensitive and does not require a '.' prefix.

For example:

```
LIST EXT(CSV)
```

SORT The optional sort keyword allows sorting on file name or modified date.

The default value is *NONE.

REVERSE The optional reverse keyword allows the sorted order to be reversed.
The default value is *NO.

Examples

RDML

This will return a list of file names in the /csvdata directory that have a CSV extension.

```
DEF_LIST NAME(#WRKLIST) FIELDS(#PATH) TYPE(*WORKING)  
USE BUILTIN(JSM_COMMAND) WITH_ARGS('LIST DIR(/CSVDATA) EXT(CSV)')
```

or equivalently,

```
DEF_LIST NAME(#WRKLIST) FIELDS(#PATH) TYPE(*WORKING)  
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET DIR(/CSVDATA)') T  
USE BUILTIN(JSM_COMMAND) WITH_ARGS('LIST EXT(CSV) SERVICE')
```

RDMLX

```
Def_list name(#wrklist) fields(#path) type(*working)  
use builtin(jsmx_command) with_args(#jsmhandle 'list dir(/csvdata) ext(csv)')
```

COMMIT

The COMMIT command commits the transaction to the database.

```
COMMIT ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('COMMIT') TO_GET(#JSI
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'commit') to_get(#jsmxsts #
```

ROLLBACK

The ROLLBACK command issues a rollback transaction to the current connection.

```
ROLLBACK ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('ROLLBACK') TO_GET(#
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'rollback') to_get(#jsmxsts :
```

DISCONNECT

The DISCONNECT command disconnects from the current connected database.

```
DISCONNECT ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('DISCONNECT') TO_GET
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'disconnect') to_get(#jsmxs
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSA Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
                *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE

- *SERVICE - read service trace file.
- *TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.message type (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html")

will be returned to the application in this case):

```
com.acme.property.messageType=html
```

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TC
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'service_unload') to_get(#js
```

5.35.4 SVFileService Examples

Examples are supplied for the most common client programming languages:

[RDML](#)

[RDMLX](#)

[RPG](#)

5.35.5 Troubleshooting

Attempting to READ a separated variable file, if you get this error message:
token count does not equal list field count at record line : nn
it indicates your working list definition does not match the information being read from the file.

There are several possible reasons:

- Regardless of whether the file is read as *CSV, *TSV or *SV the working list receiving the file information must be defined to accommodate the maximum number of columns in the content.
- The separator value must match the file information i.e. you will get an error if you attempt to read a CSV file as *TSV.
- Any errors in the file information (for example, inconsistent number of separated value on a given line of the file).

5.36 ExcelService

The ExcelService provides a means for applications to create and read Microsoft Excel documents. Since Microsoft Excel is so widely used, Excel documents can be a convenient way to exchange data between trading units or partners. An application may also use the service for extracting subsets of data from a corporate database for further analysis or presentation in an Excel workbook.

This service uses the open source Apache POI classes. Refer to <http://poi.apache.org/spreadsheet/>.

Due to the dependence on Apache POI, the ExcelService requires JDK 1.5 or higher.

The Apache POI and ExcelService jar files are shipped as extra and can be downloaded from <http://www.lansa.com/support> by searching for ExcelService.

When adding images to a sheet, only PNG and JPEG formats are supported.

Related Services

The ExcelService is not dependent on other services.

Technical Specifications

5.36.1 What can I use the ExcelService for?

The service provides support for multiple worksheets in a workbook. An application can interrogate the service to find the names of the worksheets in a workbook and then individually process the data in all or selected worksheets. Data can be read from or written to a worksheet in specified row/column ranges using LANSAR DML working lists or 3GL data structures.

The following paragraph provides an example of how ExcelService could be used:

Processing Monthly Timesheets

Employees record their billable information in a preformatted Excel spreadsheet. At the end of the month each employee emails a copy of their timesheet to the Head Office to facilitate the billing of clients.

An application at Head Office is used to pick up and process the emails from the agreed mailbox – this could potentially use the POP3MailService. The Excel document associated with each email is processed using the ExcelReadService into a database file. When all timesheets have been received, an internal application generates the appropriate invoices for each client.

5.36.2 Using the ExcelService

Typical ExcelService Command Usage

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application that reads an EXCEL spreadsheet would typically issue the following sequence of commands:

JSM(X)_OPEN

JSM(X)_COMMANDs

SERVICE_LOAD

OPEN

READ / WRITE

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Refer to [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.36.3 ExcelService Commands

Your application issues commands to the ExcelReadService by passing the command strings through the Java Service Manager using the JSM_COMMAND or JSMX_COMMAND Built-In Function, or an equivalent Built-In Function or API for your chosen development language.

The commands that the ExcelReadService processes are:

SERVICE_LOAD

OPEN

CREATE

SAVE

GET

SET

ADD

REMOVE

READ

WRITE

CLOSE

SERVICE_GET

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
---->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

SERVICE The name of the service to be loaded - in this case ExcelService.

TRACE To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.

The possible values for the TRACE keyword are:

- *NO
- *YES
- *ERROR

The trace option *ERROR will turn on tracing and if the

service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call

TRACE_NAME This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD  
SERVICE(EXCELSERVICE)'  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD)  
TO_GET(#JSMSTS #JSMMSG)
```

RDMLX

```
#jasmcommand := 'service_load service(ExcelService)'  
use builtin(jsmx_command) with_args(#jasmxhdl1 #jasmcommand)  
to_get(#jasmxsts #jasmxmsg)
```

OPEN

The OPEN command is used to open a specific Excel document for reading or writing.

Required

OPEN ----- *FILE* ----- *file path* -----|

Keywords

FILE The path and file name to be opened. The file name should include an appropriate Excel file extension, for example XLSX for an Excel Workbook.

For example:

```
use builtin(jsmx_command) with_args(#jsmhandle 'open  
file(product.xlsx)') to_get(#jstmsts #jsmmsg)
```

```
use builtin(jsmx_command) with_args(#jsmhandle 'close')  
to_get(#jstmsts #jsmmsg)
```

Examples

RDML

```
USE BUILTIN(JSM_COMMAND)  
WITH_ARGS('OPEN FILE(product.xlsx)') TO_GET(#JSMSTS #JSMMSG)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'open file(product.xlsx)')  
to_get(#jstmsts #jsmmsg)
```

CREATE

The CREATE command is used to create a new Excel document for writing. Use the SAVE command and FILE keyword to save the created document. The optional USE keyword can be used to specify a template document used to create the new document.

Required

```
CREATE ----->
```

Optional

```
>-- FORMAT ----- *XLS ----->  
          *XLSX
```

```
>-- USING ----- file path -----|
```

Keywords

FORMAT Use *XLS to the create Excel 97-2003 format.

The default *XLSX creates the XML based format.

USING The file path for an existing Excel document which is to be used to as a template to create a new document. The ExcelService determines the file format from the file path extension.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND)  
WITH_ARGS('CREATE') TO_GET(#JSMSTS #JSMMSG)  
USE BUILTIN(JSM_COMMAND)  
WITH_ARGS('CREATE FORMAT(*XLS)') TO_GET(#JSMSTS #JSMMSG)
```

or

```
USE BUILTIN(JSM_COMMAND)
```

```
WITH_ARGS('CREATE USING(template.xlsx)') TO_GET(#JSMSTS  
#JSMMSG)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'create') to_get(#jsmsts  
#jsmmsg)
```

SAVE

The SAVE command is used to save a modified document. The FILE keyword is optional. If the document has been opened then no FILE keyword is required as the original document is used for the save. Use the FILE keyword for saving created documents or saving an opened document to a different document path.

Required

SAVE ----->

Optional

>-- FILE ----- file path -----|

Keywords

FILE The path and file name to be for the saved document. The file name should include an appropriate Excel file extension, for example XLSX for an Excel Workbook.

For example:

```
use builtin(jsmx_command) with_args(#jsmhandle 'save
file(product.xlsx)') to_get(#jstmsts #jstmmsg)
```

Examples

RDML

```
USE BUILTIN(JSM_COMMAND)
WITH_ARGS('SAVE FILE(product.xlsx)') TO_GET(#JSMSTS #JSMMSG)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'save file(product.xlsx)')
to_get(#jstmsts #jstmmsg)
```

GET

The GET command is used to get information about the current open Excel document.

Required

```
GET ----- OBJECT ----- *SHEETS ----->
      *NAMES
      *ROWCOUNT
      *COLUMNCOUNT
      *CELLCOUNT
      *CELL
```

Optional

```
>-- SHEET ----- value ----->
>-- ROW ----- value ----->
>-- R1C1 ----- value ----->
>-- NAME ----- value ----->
>-- SCALE ----- value ----->
>-- NUMBERFORMAT ----- *NONE -----
|
      *DEFAULT
      *CLIENT
      *USERAGENT
```

Keywords

OBJECT

The type of object to return must be indicated as either *SHEETS, *NAMES, *ROWCOUNT, *COLUMNCOUNT, *CELLCOUNT or *CELL.

A value of *SHEETS returns a working list with all the sheet names in the current document.

A value of *NAMES returns a working list with all the defined names in the current document. If the working list has two fields then the second field receives the define name reference formula.

A value of *ROWCOUNT returns the number of rows in the worksheet specified by the SHEET keyword into the #JSMMSG field.

A value of *COLUMNOUNT returns the maximum number of columns in the worksheet specified by the SHEET keyword into the #JSMMSG field.

A value of *CELLCOUNT returns the number of cells in the worksheet specified by the SHEET keyword and the row specified by the ROW keyword into the #JSMMSG field.

A value of *CELL returns the cell value into the #JSMMSG response field. Refer to keywords R1C1, NAME, SCALE and NUMBERFORMAT.

SHEET	A specific sheet name can be entered otherwise the current context sheet is used.
R1C1	Cell row and column. This keyword can be used instead of NAME.
NAME	Defined name. The defined name contains the cell location in standard Excel A1 reference style. For example. C10 or SheetName!C10.
SCALE	This optional keyword specifies the decimal scale that will be used on numeric cells that have a general number format. The default value is 4.
NUMBERFORMAT	This optional keyword handles numeric strings where the decimal separator is not the decimal point character ".". Refer to NUMBERFORMAT for more information.

Examples

The following examples use the GET command to retrieve information about the current Excel document.

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET  
OBJECT(*SHEETS) SERVICE_LIST(SHEET) TO_GET(#JSMSTS  
#JSMMSG #WRKLST)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'get object(*rowcount)  
sheet(parts)') to_get(#jsmsts #jsmmsg)
```

SET

The SET command is used to change the current context sheet, sheet column widths and sheet settings of the specified sheet.

To change the context sheet, use the OBJECT keyword with a value of *CONTEXT.

Required

```
SET ----->
  >-- OBJECT ----- *CONTEXT----->
  >-- SHEET ----- value -----|
```

Keywords

OBJECT *CONTEXT

SHEET A specific sheet name that must exist in the document.

To change the sheet columnwidths, use the OBJECT keyword with a value of *COLUMNWIDTH.

Required

```
SET ----->
  >-- OBJECT ----- *COLUMNWIDTH----->
>
```

Optional

```
>-- SHEET ----- value ----->
>-- WIDTH ----- value ----->
>-- RANGE ----- n,n -----|
```



```

>-- PROTECT ----- *YES ----->
                *NO

>-- PASSWORD ----- value ----->

>-- FREEZE ----- value ----->

>-- MARGIN ----- value ----->

>-- HEADER ----- *LEFT ----->
                *RIGHT
                *CENTER

>-- FOOTER ----- *LEFT ----->
                *RIGHT
                *CENTER

>-- CONTENT ----- value -----|

```

Keywords

OBJECT	*SHEET
SHEET	A specific sheet name can be entered otherwise the current context sheet is used.
SELECTED	Set this sheet to be the selected sheet. Possible values are *YES or *NO.
DISPLAYGRID	Show grid. Possible values are *YES or *NO.
PRINTGRID	Print grid. Possible values are *YES or *NO.
PRINTAREA	Set print area. Example value: \$A\$1:\$B\$2.
PRINTOFIT	Enable or disable print to fit. Possible values are *YES or *NO.

PROTECT	Enable or disable sheet protection. Possible values are *YES or *NO.
PASSWORD	The worksheet protection password is optional and is used in combination with the PROTECT keyword.
FREEZE	Create a split freeze pane. Use the value of n,n to create a column split,row split freeze pane. Use the value of n,n,n,n to create column split,row split,left most column, top row freeze pane. Use the value of 0,0 to remove the split freeze pane.
MARGIN	Set sheet margins. The value is top margin, bottom margin, left margin, right margin, header margin and footer margin component values comma separated. An empty string value is used to allow a component value to be ignored.
HEADER	Specify which header content to change. The possible values are *LEFT,*RIGHT or *CENTER.
FOOTER	Specify which footer content to change. The possible values are *LEFT, *RIGHT or *CENTER.
CONTENT	The header or footer content text. Special substitution values can be used within the header and footer text. &D Date &T Time &P Page Number &N Total Number of Pages &B Bold &U Underline &I Italics &S Strike Through &E Double Underline &X Superscript &Y Subscript &F Workbook Name

&AWorksheet Name

&"font name" Font Name

&nn Font Size (Must be a two digit number. 01 to 99)

&NL New Line

ADD

The ADD command is used to add sheets, cell styles, named areas, images, formulas, hyperlinks, comments and cell merges to an Excel document.

To add a sheet, use the OBJECT keyword with a value of *SHEET.

Required

```
ADD ----- OBJECT ----- *SHEET ----->
```

```
>-- SHEET ----- value -----|
```

Keywords

OBJECT *SHEET

SHEET A new sheet name.

To add a comment, use the OBJECT keyword with a value of *COMMENT.

Required

```
ADD ----- OBJECT ----- *COMMENT -----
```

```
->
```

```
>-- R1C1 ----- n,n -----|
```

Optional

```
>-- SHEET ----- value ----->
```

```
>-- AREA ----- n,n ----->
```

```
>-- COMMENT ----- value -----|
```

Keywords

OBJECT *COMMENT

- SHEET A specific sheet name can be entered otherwise the current context sheet is used.
- R1C1 Cell row and column.
- AREA Display rectangle.
 Default value is 2,4.
- COMMENT The comment text. If the comment keyword is not used, then the cell comment is removed.

To merge cells, use the OBJECT keyword with a value of *CELLMERGE.

```

                                Required

ADD ----- OBJECT ----- *CELLMERGE -----
->

    >-- R1C1 ----- n,n -----|
    >-- R2C2 ----- n,n -----|

                                Optional

    >-- SHEET ----- value ----->
```

Keywords

- OBJECT *CELLMERGE
- SHEET A specific sheet name can be entered otherwise the current context sheet is used.
- R1C1 Cell row and column.
- R2C2 Cell row and column.

To add an image, use the OBJECT keyword with a value of *IMAGE.

```

                                Required
```

```

ADD ----- OBJECT ----- *IMAGE ----->

>-- FILE ----- value ----->

>-- R1C1 ----- n,n ----->

                                Optional

>-- SHEET ----- value ----->

>-- RESIZE ----- n ----->

>-- ANCHOR ----- *MOVE -----|
                                *MOVESIZE
                                *NOMOVESIZE

```

Keywords

OBJECT *IMAGE

SHEET A specific sheet name can be entered otherwise the current context sheet is used.

FILE The path to the PNG or JPEG image file.

R1C1 Cell row and column.

RESIZE This keyword is optional and specifies image resize amount. The default value is 1.0.

ANCHOR This keywords specifies how the image moves and sizes when column is sized. The default value is *MOVESIZE.

To add a formula, use the OBJECT keyword with a value of *FORMULA.

```

                                Required

ADD ----- OBJECT ----- *FORMULA ----->
>

```

>-- *FORMULA* ----- *value* ----->

>-- *R1C1* ----- *n,n* ----->

Optional

>-- *SHEET* ----- *value* -----|

Keywords

OBJECT *FORUMLA

SHEET A specific sheet name can be entered otherwise the current context sheet is used.

FORMULA Excel formula.

R1C1 Cell row and column.

To add a hyperlink use the OBJECT keyword with a value of *HYPERLINK.

Required

ADD ----- *OBJECT* ----- **HYP**ERLINK* ----->

>-- *ADDRESS* ----- *value* ----->

>-- *R1C1* ----- *n,n* ----->

Optional

>-- *SHEET* ----- *value* ----->

>-- *TYPE* ----- **URL* ----->

**FILE*

**EMAIL*

**DOCUMENT*

>-- LABEL ----- value -----|

Keywords

OBJECT *HYPERLINK

SHEET A specific sheet name can be entered otherwise the current context sheet is used.

ADDRESS Hyperlink address.

R1C1 Cell row and column.

TYPE Type of hyperlink.

LABEL Hyperlink label.

To add a name to the workbook use the OBJECT keyword with a value of *NAME.

Required

ADD ----- OBJECT ----- *NAME ----->

>-- NAME ----- value ----->

>-- REFERENCE ----- value ----->

Optional

>-- SHEET ----- value ----->

>-- COMMENT ----- value -----|

Keywords

OBJECT *NAME

NAME Name

REFERENCE Forumla Reference.
 Example: A1:C5
 Example: Sheet1!A1:C5
 Example: SUM(Sheet1!I\$2:I\$6)

SHEET Sheet name to prefix reference, if sheet name not included in reference.
 Example: Sheet1

COMMENT Comment

To add a cell style, use the OBJECT keyword with a value of *CELLSTYLE.

Required

*ADD ----- OBJECT ----- *CELLSTYLE -----*
 >

>-- *SHEET ----- value -----*>

>-- *COLUMN ----- n -----*>
n,n

Optional

>-- *RANGE ----- n,n -----*>

>-- *TYPE ----- *NUMBER -----*>

- *DATE*
- *BOOLEAN*
- *STRING*
- *BLANK*

>-- *FORMAT----- value -----*>

- *FORMAT0*
- *FORMAT1*
- *FORMAT2*
- *FORMAT3*
- *FORMAT4*

**FORMAT5*
**FORMAT6*
**FORMAT7*
**FORMAT8*
**FORMAT9*
**FORMAT10*
**FORMAT11*
**FORMAT12*
**FORMAT13*
**FORMAT14*
**FORMAT15*
**FORMAT16*
**FORMAT17*
**FORMAT18*
**FORMAT19*
**FORMAT20*
**FORMAT21*
**FORMAT22*
**FORMAT37*
**FORMAT38*
**FORMAT39*
**FORMAT40*
**FORMAT41*
**FORMAT42*
**FORMAT43*
**FORMAT44*
**FORMAT45*
**FORMAT46*
**FORMAT47*
**FORMAT48*
**FORMAT49*

>-- *FONT* ----- *value* ----->

**ARIAL*
**CALIBRI*
**COURIER*
**COURIERNEW*
**TAHOMA*
**TIMES*

>-- *FONTSIZE* ----- *value* ----->

>-- *FONTCOLOR* ----- **AQUA* ----->

**AUTOMATIC*

**BLACK*

**BLUE*

**BLUEGREY*

**BRIGHTGREEN*

**BROWN*

**CORAL*

**CORNFLOWERBLUE*

**DARKBLUE*

**DARKGREEN*

**DARKRED*

**DARKTEAL*

**DARKYELLOW*

**GOLD*

**GREEN*

**GREY25*

**GREY40*

**GREY50*

**GREY80*

**INDIGO*

**LAVENDAR*

**LEMONCHIFFON*

**LIGHTBLUE*

**LIGHTCORNFLOWERBLUE*

**LIGHTGREEN*

**LIGHTORANGE*

**LIGHTTURQUOISE*

**LIGHTYELLOW*

**LIME*

**MAROON*

**OLIVEGREEN*

**ORANGE*

**ORCHID*

**PALEBLUE*

**PINK*

**PLUM
*RED
*ROSE
*SEAGREEN
*SKYBLUE
*TAN
*TEAL
*TURQUOISE
*VIOLET
*WHITE
YELLOW

*>-- BOLD ----- *YES ----->
NO

*>-- ITALIC ----- *YES ----->
NO

*>-- BORDER ----- *ALL ----->
*NONE
*TOP
*BOTTOM
*LEFT
RIGHT

*>-- BORDERSTYLE ----- *NONE ----->
*MEDIUM
*THICK
*THIN
*HAIR
DOUBLE

*>-- BORDERCOLOR ----- *AQUA ----->
*AUTOMATIC
*BLACK
*BLUE
*BLUEGREY
*BRIGHTGREEN
BROWN

*CORAL
*CORNFLOWERBLUE
*DARKBLUE
*DARKGREEN
*DARKRED
*DARKTEAL
*DARKYELLOW
*GOLD
*GREEN
*GREY25
*GREY40
*GREY50
*GREY80
*INDIGO
*LAVENDAR
*LEMONCHIFFON
*LIGHTBLUE
*LIGHTCORNFLOWERBLUE
*LIGHTGREEN
*LIGHTORANGE
*LIGHTTURQUOISE
*LIGHTYELLOW
*LIME
*MAROON
*OLIVEGREEN
*ORANGE
*ORCHID
*PALEBLUE
*PINK
*PLUM
*RED
*ROSE
*SEAGREEN
*SKYBLUE
*TAN
*TEAL
*TURQUOISE
*VIOLET
*WHITE

**YELLOW*

*>-- BACKGROUND ----- *AQUA ----->*

**AUTOMATIC*

**BLACK*

**BLUE*

**BLUEGREY*

**BRIGHTGREEN*

**BROWN*

**CORAL*

**CORNFLOWERBLUE*

**DARKBLUE*

**DARKGREEN*

**DARKRED*

**DARKTEAL*

**DARKYELLOW*

**GOLD*

**GREEN*

**GREY25*

**GREY40*

**GREY50*

**GREY80*

**INDIGO*

**LAVENDAR*

**LEMONCHIFFON*

**LIGHTBLUE*

**LIGHTCORNFLOWERBLUE*

**LIGHTGREEN*

**LIGHTORANGE*

**LIGHTTURQUOISE*

**LIGHTYELLOW*

**LIME*

**MAROON*

**OLIVEGREEN*

**ORANGE*

**ORCHID*

**PALEBLUE*

**PINK*

**PLUM*

**RED*
**ROSE*
**SEAGREEN*
**SKYBLUE*
**TAN*
**TEAL*
**TURQUOISE*
**VIOLET*
**WHITE*
**YELLOW*

>-- *WRAP* ----- **YES* ----->
**NO*

>-- *LOCKED* ----- **YES* ----->
**NO*

>-- *INDENT* ----- *value* ----->

>-- *HALIGN* ----- **CENTRE* ----->
**CENTER*
**FILL*
**GENERAL*
**JUSTIFY*
**LEFT*
**RIGHT*

>-- *VALIGN* ----- **CENTRE* -----|
**CENTER*
**JUSTIFY*
**TOP*
**BOTTOM*

Keywords

OBJECT

**CELLSTYLE*

SHEET

Sheet name associated with the cell style.

COLUMN

This keyword is used to specify the column number or

range of column numbers you wish your cell style to act on. You can specify a single column number or a comma-separated range.

The format of this keyword is as follows:

COLUMN(column), for example, COLUMN(3).

COLUMN(start column, end column), for example, COLUMN(3,5).

This keyword is mandatory.

RANGE

This keyword is used to define the range of rows within the specified column on which this cell style will act.

If left blank, then the entire column will be acted upon.

The format of this keyword is as follows:

RANGE(start row, end row), for example, RANGE(3,5).

This keyword is optional.

TYPE

This keyword is used to define the column type.

The possible values are:

*NUMBER

*DATE

*BOOLEAN

*STRING

*BLANK

The default value is *NUMBER

This keyword is optional.

FORMAT

Excel format to be applied to the cell value.

The default format depends on the cell type. If the cell type is *DATE the default format is the builtin format *FORMAT14 which is an internationalised date format.

The default format for all other cell types is the builtin format *FORMAT0 which is the General format.

*FORMAT0 General

*FORMAT1 0

*FORMAT2 0.00

*FORMAT3 #,##0

*FORMAT4 #,##0.00
 *FORMAT5 "\$"#,##0_);("\$"#,##0)
 *FORMAT6 "\$"#,##0_);[Red]("\$"#,##0)
 *FORMAT7 "\$"#,##0.00_);("\$"#,##0.00)
 *FORMAT8 "\$"#,##0.00_);[Red]("\$"#,##0.00)
 *FORMAT9 0%
 *FORMAT10 0.00%
 *FORMAT11 0.00E+00
 *FORMAT12 # ?/?
 *FORMAT13 # ??/??
 *FORMAT14 m/d/yy
 *FORMAT15 d-mmm-yy
 *FORMAT16 d-mmm
 *FORMAT17 mmm-yy
 *FORMAT18 h:mm AM/PM
 *FORMAT19 h:mm:ss AM/PM
 *FORMAT20 h:mm
 *FORMAT21 h:mm:ss
 *FORMAT22 m/d/yy h:mm
 *FORMAT37 #,##0_);(#,##0)
 *FORMAT38 #,##0_);[Red](#,##0)
 *FORMAT39 #,##0.00_);(#,##0.00)
 *FORMAT40 #,##0.00_);[Red](#,##0.00)
 FORMAT41 _("\$" #,##0_);_("\$"* (#,##0);_("\$"* "-
 "_);_(@_)
 FORMAT42 _(#,##0_);_(* (#,##0);_(* "-"_);_(@_)
 FORMAT43 _("\$" #,##0.00_);_("\$"* (#,##0.00);_("\$"*
 "-"?_);_(@_)
 FORMAT44 _(#,##0.00_);_(* (#,##0.00);_(* "-"??
);(@_)
 *FORMAT45 mm:ss
 *FORMAT46 [h]:mm:ss
 *FORMAT47 mm:ss.0
 *FORMAT48 ##0.0E+0
 *FORMAT49 @

FONT

Font is any valid Windows font name.

The special values of *ARIAL, *CALIBRI, *COURIER, *COURIERNEW, *TAHOMA, *TIMES are substituted

to their standard Windows font names.

FONTSIZE	The keyword is used to define the font size. The default point size is 11. This keyword is optional.
FONTCOLOR	The keyword is used to define the font color.
BOLD	This keyword is used to set the font to bold or not. The possible values for this keyword are: *NO *YES The default value is *NO This keyword is optional.
ITALIC	This keyword is used to set the font to italics or not. The possible values for this keyword are: *NO *YES The default value is *NO This keyword is optional.
BORDER	This keyword is used to define the lines of the border. The possible values are as follows: *ALL *NONE *TOP *BOTTOM *LEFT *RIGHT There is no default value.
BORDERSTYLE	This keyword is used to define the borderstyle. The BORDER keyword needs to be specified in conjunction with this keyword.

Possible values for this keyword are as follows:

*NONE

*MEDIUM

*THICK

*THIN

*HAIR

*DOUBLE

The default value is *THIN

This keyword is optional.

BORDERCOLOR This keyword is used to define the border color.
The **BORDER** keyword needs to be specified in conjunction with this keyword.

BACKGROUND This keyword is used to define the background color.
There is no default value.

WRAP This keyword is used to enable text wrap.
The possible values for this keyword are:
*NO
*YES
The default value is *NO
This keyword is optional.

LOCKED This keyword is used to enable locked.
The possible values for this keyword are:
*NO
*YES
The default value is *NO
This keyword is optional.

INDENT This keyword is used to specify the cell value indent.
The default value is 0.
This keyword is optional.

HALIGN This keyword is used to define the horizontal alignment.

The possible values for this keyword are as follows:

*CENTRE

*CENTER

*FILL

*GENERAL

*JUSTIFY

*LEFT

*RIGHT

There is no default value.

This keyword is optional.

VALIGN

This keyword is used to define the vertical alignment.

The possible values for this keyword are as follows:

*CENTRE

*CENTER

*JUSTIFY

*TOP

*BOTTOM

There is no default value.

This keyword is optional.

REMOVE

The REMOVE command is used to remove a name, sheet, sheet cell styles, print area, rows and columns.

To remove a name, use the OBJECT keyword with a value of *NAME.

```

                                     Required
REMOVE ---- OBJECT ----- *NAME -----
>
    >-- NAME ----- value -----|
```

Keywords

OBJECT *NAME

NAME Name.

To remove a sheet, use the OBJECT keyword with a value of *SHEET.

```

                                     Required
REMOVE ---- OBJECT ----- *SHEET -----
>
                                     Optional
    >-- SHEET ----- value -----|
```

Keywords

OBJECT *SHEET

SHEET A specific sheet name can be entered otherwise the current context sheet is used.

To remove added sheet cell styles, use the OBJECT keyword with a value of *CELLSTYLE.

Required

REMOVE ----- *OBJECT* ----- **CELLSTYLE* -----
-->

Optional

>-- *SHEET* ----- *value* -----|
 **ALL*

Keywords

OBJECT **CELLSTYLE*

SHEET A specific sheet name can be entered otherwise the current context sheet is used.

The special value of **ALL* will clear all added cell styles for all sheets.

To remove a sheet print area, use the **OBJECT** keyword with a value of **PRINTAREA*.

Required

REMOVE ----- *OBJECT* ----- **PRINTAREA* -----
-->

Optional

>-- *SHEET* ----- *value* -----|

Keywords

OBJECT **PRINTAREA*

SHEET A specific sheet name can be entered otherwise the current context sheet is used.

To remove rows, use the OBJECT keyword with a value of *ROW.

```
                                Required
REMOVE ---- OBJECT ----- *ROW -----
>
    >-- RANGE ----- n,n ----->
                                Optional
    >-- SHEET ----- value -----|
```

Keywords

OBJECT *ROW

SHEET A specific sheet name can be entered otherwise the current context sheet is used.

RANGE Range of rows. A single row number or two comma separated row numbers to specify a range of rows.

To remove columns, use the OBJECT keyword with a value of *COLUMN.

```
                                Required
REMOVE ---- OBJECT ----- *COLUMN -----
-->
    >-- RANGE ----- n,n ----->
                                Optional
    >-- SHEET ----- value -----|
```

Keywords

OBJECT *COLUMN

SHEET A specific sheet name can be entered otherwise the current context sheet is used.

RANGE Range of columns. A single column number or two comma separated column numbers to specify a range of columns.

READ

The READ command is used to read a range of cells from the nominated worksheet into a working list. Each cell value has trailing blanks trimmed.

READ

Required

Optional

```
>-- SHEET ----- value ----->
>-- R1C1 ----- n,n ----->
>-- ROWCOUNT ----- value ----->
>-- IGNORE ----- *NONE ----->
      *EMPTY
>-- SCALE ----- value ----->
>-- NUMBERFORMAT --- *NONE -----|
      *DEFAULT
      *CLIENT
      *USERAGENT
      value
```

Keywords

SHEET	A specific sheet name can be entered otherwise the current context sheet is used.
R1C1	The starting row and column defaults to 1, 1. This indicates the starting position for rows and columns reading.
ROWCOUNT	Specify the number of rows to read. If no ROWCOUNT keyword is used all rows to the end of the sheet are read.

IGNORE	The default value of *NONE indicates that all entries should be processed. The alternative option *EMPTY indicates that empty entries should be ignored. An empty entry is where all cell values that make a list entry have an empty string value.
SCALE	This optional keyword specifies the decimal scale that will be used on numeric cells that have a general number format. The default value is 4.
NUMBERFORMAT	This optional keyword handles numeric strings where the decimal separator is not the decimal point character ".". Refer to NUMBERFORMAT for more information.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('READ SHEET(PARTS)
SERVICE_LIST(LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
TO_GET(#JSMSTS #JSMMSG #ORDLIST)
```

or

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('READ SHEET(PARTS)
R1C1(10,3)
SERVICE_LIST(LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
TO_GET(#JSMSTS #JSMMSG #ORDLIST)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'read sheet(parts)')
to_get(#jsmsts #jsmmsg #ordlist)
```

or

```
use builtin(jsmx_command) with_args(#jsmhandle 'read sheet(parts)
r1c1(10,3)') to_get(#jsmsts #jsmmsg #ordlist)
```


WRITE

The WRITE command is used to write a range of cells from a working list to a nominated worksheet.

All rows and columns from the working list are written out to the specified sheet using the R1C1 start position. Any fields specified in the EXCLUDE keyword are not included in the write operation.

WRITE

Required

Optional

>-- SHEET ----- value ----->

>-- R1C1 ----- n,n ----->

>-- EXCLUDE ----- field,field -----|

Keywords

- SHEET** A specific sheet name can be entered otherwise the current context sheet is used.
- R1C1** The starting row and column defaults to 1, 1. This indicates the starting position for rows and columns from the working list to be written out to the specified sheet.
- EXCLUDE** An optional comma-separated list of working list fields. These fields will be excluded from the write operation.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('WRITE SHEET(PARTS)
R1C1(10,3)
SERVICE_LIST(LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
TO_GET(#JSMSTS #JSMMSG #ORDLIST)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'write sheet(parts)
r1r1(10,3)') to_get(#jsmsts #jsmmsg #ordlist)
```

CLOSE

The CLOSE command closes the opened or created workbook.

CLOSE ----- no keywords -----|

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(CLOSE)  
TO_GET(#JSMSTS #JSMMSG)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 close) to_get(#jsmxsts  
#jsmxmsg)
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values. When used in this way, there is a possibility of conflict between application-defined keyword names and those used by LANSAs Integrator. To reduce the possibility of conflict, your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
        *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE *SERVICE - read service trace file.
*TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.messageType (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html" will be returned to the application in this case):

com.acme.property.messageType=html

RDML

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'

RDMLX

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD)  
TO_GET(#JSMSTS #JSMMSG)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhde1 service_unload)  
to_get(#jsmxsts #jsmxmsg)
```

5.37 ExcelReadService

The ExcelReadService provides a means for applications to create and read Microsoft Excel documents. Since Microsoft Excel is so widely used, Excel documents can be a convenient way to exchange data between trading units or partners. An application may also use the service for extracting subsets of data from a corporate database for further analysis or presentation in an Excel workbook.

This service uses the open source Java Excel API classes. Refer to <http://jexcelapi.sourceforge.net>.

When adding images to a sheet, only the Portable Network Graphics (PNG) format is supported.

Related Services

The ExcelReadService is not dependent on other services.

Technical Specifications

The following service properties control the locale, region and language values used when reading and writing workbooks:

- excel.locale
- excel.region
- excel.language

5.37.1 What can I use the ExcelReadService for?

The service provides support for multiple worksheets in a workbook. An application can interrogate the service to find the names of the worksheets in a workbook and then individually process the data in all or selected worksheets. Data can be read from or written to a worksheet in specified row/column ranges using LANSAR DML working lists or 3GL data structures.

The following paragraph provides an example of how ExcelReadService could be used:

Processing Monthly Timesheets

Employees record their billable information in a preformatted Excel spreadsheet. At the end of the month each employee emails a copy of their timesheet to the Head Office to facilitate the billing of clients.

An application at Head Office is used to pick up and process the emails from the agreed mailbox – this could potentially use the POP3MailService. The Excel document associated with each email is processed using the ExcelReadService into a database file. When all timesheets have been received, an internal application generates the appropriate invoices for each client.

5.37.2 Using the ExcelReadService

Typical ExcelReadService Command Usage

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application that reads an EXCEL spreadsheet would typically issue the following sequence of commands:

JSM(X)_OPEN

JSM(X)_COMMANDs

SERVICE_LOAD

OPEN

READ / WRITE

CLOSE

SERVICE_UNLOAD

JSM(X)_CLOSE

Refer to [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.37.3 ExcelReadService Commands

Your application issues commands to the ExcelReadService by passing the command strings through the Java Service Manager using the JSM_COMMAND or JSMX_COMMAND Built-In Function, or an equivalent Built-In Function or API for your chosen development language.

The commands that the ExcelReadService processes are:

SERVICE_LOAD

OPEN

GET

SET

ADD

REMOVE

READ

WRITE

DEFINE

CLOSE

SERVICE_GET

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
---->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

- | | |
|---------|--|
| SERVICE | The name of the service to be loaded - in this case ExcelReadService. |
| TRACE | To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.
The possible values for the TRACE keyword are: <ul style="list-style-type: none">• *NO• *YES• *ERROR The trace option *ERROR will turn on tracing and if the |

service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call

TRACE_NAME This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD  
SERVICE(EXCELREADSERVICE)'  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD)  
TO_GET(#JSMSTS #JSMMSG)
```

RDMLX

```
#jasmcommand := 'service_load service(EXCELREADSERVICE)'  
use builtin(jsmx_command) with_args(#jasmxhdl1 #jasmcommand)  
to_get(#jasmxsts #jasmxmsg)
```

OPEN

The OPEN command is used to open a specific Excel document for reading or writing.

To change the mode after the Excel document has been opened:

1. Use the CLOSE command, then
2. Re-open the document with the appropriate mode.

An OPEN command should be issued before using the GET, READ or WRITE commands.

Required

OPEN ----- *FILE* ----- *file path* ----->

Optional

>-- *MODE* ----- **READ* ----->

**WRITE*

>-- *REPLACE* ----- **NO* ----->

**YES*

>-- *TEMPLATE* ----- *file path* -----|

Keywords

FILE The path and file name to be opened or created. The file name should include an appropriate Excel file extension, for example XLS for an Excel Workbook or XLT for an Excel Template.

A new Excel document can be created by simply using the OPEN command followed by the CLOSE command.

For example:

```
use builtin(jsmx_command) with_args(#jsmhandle 'open
file(product.xls) mode(*write) template(product.xlt)')
to_get(#jsmsts #jsmmsg)
```

```
use builtin(jsmx_command) with_args(#jsmhandle 'close')
to_get(#jsmsts #jsmmsg)
```

- MODE** File open mode can be set as *READ or *WRITE. The document must be opened with the appropriate mode to before using the READ or WRITE command. The default value is *READ.
- REPLACE** Indicate whether the current version of the file is to be replaced. The default value is *NO. This is used in conjunction with MODE(*WRITE).
- TEMPLATE** The file path for an Excel document which is to be used to create a document with the name nominated in the FILE keyword.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND)
WITH_ARGS('OPEN FILE(product.xls)') TO_GET(#JSMSTS #JSMMSG)
```

or

```
USE BUILTIN(JSM_COMMAND)
WITH_ARGS('OPEN FILE(product.xls) MODE(*WRITE) REPLACE(*YES)
TO_GET(#JSMSTS #JSMMSG)
```

or

```
USE BUILTIN(JSM_COMMAND)
WITH_ARGS('OPEN FILE(product.xls) MODE(*WRITE) REPLACE(*YES)
template.xlt)') TO_GET(#JSMSTS #JSMMSG)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'open file(product.xls)')
to_get(#jsmsts #jsmmsg)
```


GET

The GET command is used to get information about the current open Excel document.

The GET command must be preceded by an OPEN command with read mode.

Required

```
GET ----- OBJECT ----- *SHEETS ----->
                        *ROWCOUNT
```

Optional

```
>-- SHEET ----- sheet1 -----|
                        value
```

Keywords

OBJECT The type of object to return must be indicated as either *SHEETS or *ROWCOUNT.

A value of *SHEETS returns a working list with all the sheet names in the current document.

A value of *ROWCOUNT returns the number of rows in the worksheet specified by the SHEET keyword in the JSM message field.

SHEET This keyword is used in combination with the value OBJECT(*ROWCOUNT) to indicate which sheet the row count should be returned for.

A specific sheet name can be entered otherwise the default value of sheet1 will be used.

Examples

The following examples use the GET command to retrieve information about the current Excel document.

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET  
OBJECT(*SHEETS) SERVICE_LIST(SHEET) TO_GET(#JSMSTS  
#JSMMSG #WRKLST)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'get object(*rowcount)  
sheet(parts)') to_get(#jsmsts #jsmmsg)
```

SET

The SET command is used to change the sheet settings of the specified sheet.
The SET command must be preceded by an OPEN command with write mode.

Required

SET ----->

Optional

>-- *SHEET* ----- *sheet1* ----->
value

>-- *FREEZE* ----- *value* ----->

>-- *SELECTED* ----- **YES* ----->
**NO*

>-- *GRIDS* ----- **YES* ----->
**NO*

>-- *ORIENTATION* ----- **PORTRAIT* ----->
**LANDSCAPE*

>-- *PROTECT* ----- **YES* ----->
**NO*

>-- *PASSWORD* ----- *value* ----->
**NONE*

>-- *MARGIN* ----- *value* ----->

>-- *HEADER* ----- **LEFT* ----->
**RIGHT*
**CENTER*

>-- *FOOTER* ----- **LEFT* ----->

**RIGHT*
**CENTER*

>-- *CONTENT* ----- *value* ----->

>-- *PRINTAREA* ----- *value* ----->

>-- *PRINTTOFIT* ----- **YES* ----->
**NO*

>-- *PRINTSCALE* ----- *value* -----|

Keywords

SHEET	A specific sheet name can be entered otherwise the default value of sheet1 will be used.
FREEZE	Set horizontal row freeze or column vertical freeze. The value is the horizontal row and the vertical column component values comma separated. An empty string value is used to allow a component value to be ignored.
SELECTED	Set this sheet to be the selected sheet. Possible values are <i>*YES</i> or <i>*NO</i> .
GRIDS	Show or hide the sheet grids. Possible values are <i>*YES</i> or <i>*NO</i> .
ORIENTATION	Specify the sheet orientation. Possible values are <i>*LANDSCAPE</i> or <i>*PORTRAIT</i> .
PROTECT	Enable or disable sheet protection.
PASSWORD	The worksheet protection password is optional and is used in combination with the PROTECT keyword. The special value of <i>*NONE</i> causes the password to be removed.
MARGIN	Set sheet margins. The value is top margin, bottom margin, left margin, right margin, header margin and footer margin component values comma separated.

	An empty string value is used to allow a component value to be ignored. The unit of measurement is in inches as a float value.
HEADER	Specify which header content to change. The possible values are *LEFT,*RIGHT or *CENTER.
FOOTER	Specify which footer content to change. The possible values are *LEFT, *RIGHT or *CENTER.
CONTENT	The header or footer content text. Special substitution values can be used within the header and footer text. &D Date &T Time &P Page Number &N Total Number of Pages &B Bold &U Underline &I Italics &S Strike Through &E Double Underline &X Superscript &Y Subscript &F Workbook Name &AWorksheet Name &"font name" Font Name &nn Font Size (Must be a two digit number. 01 to 99) &NL New Line
PRINTAREA	Set print area. The value is a cell range specified by comma separated values row1, column1, row2, column2.
PRINTTOFIT	Enable or disable fits to page mode. The possible values are *YES or *NO.
PRINTSCALE	Set print scale factor.

ADD

The ADD command is used to add images, formulas and hyperlinks to an Excel document.

The ADD command must be preceded by an OPEN command with write mode.

Required

```
ADD ----- OBJECT ----- *IMAGE ----->
                        *FORMULA
                        *HYPERLINK
```

Optional

```
>-- SHEET ----- sheet1 -----|
                        value
```

Keywords

OBJECT The type of object to add to the document *IMAGE, *FORMULA, or *HYPERLINK.

A value of *IMAGE is used to add a image to a specified sheet.

A value of *FORMULA is used to add a formula to a specified sheet.

A value of *HYPERLINK is used to add a .hyperlink to a specified sheet.

SHEET A specific sheet name can be entered otherwise the default value of sheet1 will be used.

Required

```
ADD ----- OBJECT ----- *IMAGE ----->
>-- FILE ----- value ----->
```

```
>-- R1C1 ----- n,n ----->
```

Optional

```
>-- R2C2 ----- n,n ----->
```

```
>-- ANCHOR ----- *MOVE -----|  
      *MOVESIZE  
      *NOMOVESIZE
```

Keywords

OBJECT *IMAGE to add an image to the specified sheet.

SHEET A specific sheet name can be entered otherwise the default value of sheet1 will be used.

FILE The path to the PNG image file.

R1C1 The top-left cell where the image is added.

R2C2 This keyword is optional and specifies the bottom-right cell of the rectangle formed with the value of R1C1.
If not value is specified then the images spans one cell.

ANCHOR This keywords specifies how the image moves and sizes when column is sized. The default value is *MOVESIZE.

Required

```
ADD ----- OBJECT ----- *FORMULA -----  
>
```

```
>-- FORMULA ----- value ----->
```

```
>-- R1C1 ----- n,n -----|
```

Keywords

OBJECT *FORUMLA to add a formula to the specified sheet.

SHEET A specific sheet name can be entered otherwise the default value of sheet1 will be used.

FORMULA Excel formula.

R1C1 The cell where the formula is to be added.
If a cell format is defined, then this format is used when creating the cell formula.

Required

ADD ----- OBJECT ----- *HYPERLINK -----
>

>-- URL ----- value ----->

>-- R1C1 ----- n,n ----->

Optional

>-- R2C2 ----- n,n ----->

>-- LABEL ----- value -----|

Keywords

OBJECT *HYPERLINK to add a hyperlink to the specified sheet.

SHEET A specific sheet name can be entered otherwise the default value of sheet1 will be used.

URL The hyperlink destination.

R1C1 The top-left cell where the hyperlink is added.

R2C2 This keyword is optional and specifies the bottom-right cell of the rectangle formed with the value of R1C1.
If not value is specified then the hyperlink spans one cell.

LABEL Hyperlink description.

Required

ADD ----- *OBJECT* ----- **HYPERLINK* -----
>

>-- *R1C1* ----- *n,n* ----->

>-- *SHEET2* ----- *value* ----->

>-- *R2C2* ----- *n,n* ----->

Optional

>-- *LABEL* ----- *value* -----|

Keywords

OBJECT **HYPERLINK* to add a hyperlink to the specified sheet.

SHEET A specific sheet name can be entered otherwise the default value of sheet1 will be used.

R1C1 The cell where the hyperlink is added.

SHEET2 The hyperlink sheet destination.

R2C2 The hyperlink cell destination.

LABEL Hyperlink description.

REMOVE

The REMOVE command is used to remove rows or columns from the Excel document.

The REMOVE command must be preceded by an OPEN command with write mode.

```

                                     Required
REMOVE ---- OBJECT ----- *ROW -----
>
                                     *COLUMN
>-- RANGE ----- n,n ----->
                                     Optional
>-- SHEET ----- sheet1 -----|
                                     value
```

Keywords

OBJECT The type of object to be removed from the document *ROW, or *COLUMN.

A value of *ROW is used to remove a single row or a range of rows from the specified sheet.

A value of *COLUMN is used to remove a single column or a range of columns from the specified sheet.

RANGE The range is either a single integer value or a comma separated pair of integer values.

SHEET A specific sheet name can be entered otherwise the default value of sheet1 will be used.

READ

The READ command is used to read a range of cells from the nominated worksheet on the currently open document into a working list. Each cell value has trailing blanks trimmed.

The READ command must be preceded by an OPEN command with read mode.

```
READ ----- SHEET ----- sheet1 ----->
                        value

>-- R1C1 ----- 1,1 ----->
                        n,n

>-- R2C2 ----- 0,0 ----->
                        n,n

>-- IGNORE ----- *NONE ----->
                        *EMPTY

>-- SCALE ----- value ----->

>-- NUMBERFORMAT --- *NONE ----->
|
                        *DEFAULT
                        *CLIENT
                        *USERAGENT
                        value
```

Keywords

SHEET	A specific sheet name can be entered or the default value of sheet1 will be used.
R1C1	Enter a specific row and column to be read. If a value is not specified the default 1,1 is used.
R2C2	The R2C2 keyword value can have a row and column value of 0. Enter a specific end row and column for read

processing. If a value is not specified the default 0,0 is used which indicates to read to the end of the sheet.

Note: Currently the column value of the R2C2 keyword is ignored and should have a value of 0 or not specified.

IGNORE

The default value of *NONE indicates that all entries should be processed. The alternative option *EMPTY indicates that empty entries should be ignored.

An empty entry is where all cell values that make a list entry have an empty string value.

SCALE

This optional keyword specifies the decimal scale that will be used on numeric cells that have a general number format. The default value is 4.

NUMBERFORMAT This optional keyword handles numeric strings where the decimal separator is not the decimal point character ".".

Refer to [NUMBERFORMAT](#) for more information.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('READ SHEET(PARTS)
SERVICE_LIST(LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
TO_GET(#JSMSTS #JSMMSG #ORDLIST)
```

or

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('READ SHEET(PARTS)
R1C1(10,3) R2C2(20,0)
SERVICE_LIST(LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
TO_GET(#JSMSTS #JSMMSG #ORDLIST)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'read sheet(parts)')
to_get(#jsmsts #jsmmsg #ordlist)
```

or

```
use builtin(jsmx_command) with_args(#jsmhandle 'read sheet(parts)
r1c1(10,3) R2C2(20,0)') to_get(#jsmsts #jsmmsg #ordlist)
```

WRITE

The WRITE command is used to write a range of cells from a working list into the nominated worksheet on the currently open document.

All rows and columns from the working list are written out to the specified sheet using the R1C1 start position. Any fields specified in the EXCLUDE keyword are not included in the write operation.

The WRITE command must be preceded by an OPEN command with write mode.

```
WRITE ----- SHEET ----- sheet1 ----->
                        value

>-- R1C1 ----- 1,1 ----->
                        n,n

>-- EXCLUDE ----- field,field ----->

>-- NUMBERFORMAT ---- *NONE -----
|
                        *DEFAULT
                        *CLIENT
                        *USERAGENT
                        value
```

Keywords

SHEET	The name of the sheet working list value will be written to. If a sheet value is not specified the default sheet1 is applied.
R1C1	The starting row and column defaults to 1, 1. This indicates the starting position for rows and columns from the working list to be written out to the specified sheet.
EXCLUDE	An optional comma-separated list of working list fields. These fields will be excluded from the write operation.

NUMBERFORMAT This optional keyword handles numeric strings where the decimal separator is not the decimal point character ".".

Refer to [NUMBERFORMAT](#) for more information.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('WRITE SHEET(PARTS)
R1C1(10,3)
SERVICE_LIST(LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
TO_GET(#JSMSTS #JSMMSG #ORDLIST)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmhandle 'write sheet(parts)
r1r1(10,3)') to_get(#jsmsts #jsmmsg #ordlist)
```

DEFINE

The DEFINE command is used to define cell formats and column widths to be used by the WRITE command.

Required

DEFINE ----- OBJECT ----- *COLUMNVIEW -----
----->

*CELLFORMAT

COLUMN ----- n ----->
n,n

Conditional

*If the OBJECT keyword is set to *COLUMNVIEW*

CLEAR ----- *NO ----->
*YES

WIDTH ----- value ----->

*If the OBJECT keyword is set to *CELLFORMAT*

CLEAR ----- *NO ----->
*YES

RANGE ----- value ----->

TYPE ----- *NUMBER ----->
*DATE
*BOOLEAN
*STRING
*BLANK

FORMAT----->

*Cell formats for TYPE *NUMBER*

**ACCOUNTINGFLOAT
*ACCOUNTINGREDFLOAT
*ACCOUNTINGINTEGER
*ACCOUNTINGREDINTEGER
*DEFAULT
*EXPONENTIAL
*FLOAT
*FORMAT1
*FORMAT2
*FORMAT3
*FORMAT4
*FORMAT5
*FORMAT6
*FORMAT7
*FORMAT8
*FORMAT9
*FORMAT10
*FRACTIONONEDIGIT
*FRACTIONTWO DIGITS
*INTEGER
*PERCENTFLOAT
*PERCENTINTEGER
*TEXT
*THOUSANDSFLOAT
*THOUSANDSINTEGER
Any valid NumberFormat - i.e. #.##*

*Cell formats for TYPE *DATE*

**DEFAULT
*FORMAT1
*FORMAT2
*FORMAT3
*FORMAT4
*FORMAT5
*FORMAT6
*FORMAT7
FORMAT8

*FORMAT9
*FORMAT10
*FORMAT11
*FORMAT12
Any valid DateFormat i.e. dd MM yyyy

COMPLEXFORMAT---- value ----->

FONT ----- *ARIAL ----->

*COURIER
*TAHOMA
*TIMES

FontColor ----- *AQUA ----->

*AUTOMATIC
*BLACK
*BLUE
*BLUEGRAY
*BLUE2
*BRIGHTGREEN
*BROWN
*CORAL
*DARKBLUE
*DARKBLUE2
*DARKGREEN
*DARKPURPLE
*DARKRED
*DARKRED2
*DARKTEAL
*DARKYELLOW
*DEFAULTBACKGROUND
*DEFAULTBACKGROUND1
*GOLD
*GRAY25
*GRAY50
*GRAY80
*GREEN
*GRAY25
*GRAY50

*GRAY80
*ICEBLUE
*INDIGO
*IVORY
*LAVENDER
*LIGHTBLUE
*LIGHTGREEN
*LIGHTORANGE
*LIGHTTURQUOISE
*LIGHTTURQUOISE2
*LIME
*OCEANBLUE
*OLIVEGREEN
*ORANGE
*PALEBLUE
*PALETTEBLACK
*PERIWINKLE
*PINK
*PINK2
*PLUM
*PLUM2
*RED
*ROSE
*SEAGREEN
*SKYBLUE
*TAN
*TEAL
*TEAL2
*TURQUOISE
*TURQUOISE2
*VERYLIGHTYELLOW
*VIOLET
*VIOLET2
*WHITE
*YELLOW
*YELLOW2

FONTSIZE ----- value ----->

*BORDER ----- *ALL ----->*

**NONE*

**TOP*

**BOTTOM*

**LEFT*

**RIGHT*

*BORDERSTYLE ----- *NONE ----->*

**MEDIUM*

**THICK*

**THIN*

**HAIR*

**DOUBLE*

*BORDERCOLOR ----- *AQUA ----->*

**AUTOMATIC*

**BLACK*

**BLUE*

**BLUEGRAY*

**BLUE2*

**BRIGHTGREEN*

**BROWN*

**CORAL*

**DARKBLUE*

**DARKBLUE2*

**DARKGREEN*

**DARKPURPLE*

**DARKRED*

**DARKRED2*

**DARKTEAL*

**DARKYELLOW*

**DEFAULTBACKGROUND*

**DEFAULTBACKGROUND1*

**GOLD*

**GRAY25*

**GRAY50*

**GRAY80*

**GREEN*

**GRAY25*

*GRAY50
*GRAY80
*ICEBLUE
*INDIGO
*IVORY
*LAVENDER
*LIGHTBLUE
*LIGHTGREEN
*LIGHTORANGE
*LIGHTTURQUOISE
*LIGHTTURQUOISE2
*LIME
*OCEANBLUE
*OLIVEGREEN
*ORANGE
*PALEBLUE
*PALETTEBLACK
*PERIWINKLE
*PINK
*PINK2
*PLUM
*PLUM2
*RED
*ROSE
*SEAGREEN
*SKYBLUE
*TAN
*TEAL
*TEAL2
*TURQUOISE
*TURQUOISE2
*VERYLIGHTYELLOW
*VIOLET
*VIOLET2
*WHITE
*YELLOW
*YELLOW2

BACKGROUND ----- *AQUA ----->

**AUTOMATIC*
**BLACK*
**BLUE*
**BLUEGRAY*
**BLUE2*
**BRIGHTGREEN*
**BROWN*
**CORAL*
**DARKBLUE*
**DARKBLUE2*
**DARKGREEN*
**DARKPURPLE*
**DARKRED*
**DARKRED2*
**DARKTEAL*
**DARKYELLOW*
**DEFAULTBACKGROUND*
**DEFAULTBACKGROUND1*
**GOLD*
**GRAY25*
**GRAY50*
**GRAY80*
**GREEN*
**GRAY25*
**GRAY50*
**GRAY80*
**ICEBLUE*
**INDIGO*
**IVORY*
**LAVENDER*
**LIGHTBLUE*
**LIGHTGREEN*
**LIGHTORANGE*
**LIGHTTURQUOISE*
**LIGHTTURQUOISE2*
**LIME*
**OCEANBLUE*
**OLIVEGREEN*
**ORANGE*

**PALEBLUE*
**PALETTEBLACK*
**PERIWINKLE*
**PINK*
**PINK2*
**PLUM*
**PLUM2*
**RED*
**ROSE*
**SEAGREEN*
**SKYBLUE*
**TAN*
**TEAL*
**TEAL2*
**TURQUOISE*
**TURQUOISE2*
**VERYLIGHTYELLOW*
**VIOLET*
**VIOLET2*
**WHITE*
**YELLOW*
**YELLOW2*

*HALIGN ----- *CENTRE ----->*
**CENTER*
**FILL*
**GENERAL*
**JUSTIFY*
**LEFT*
**RIGHT*

*VALIGN ----- *CENTRE ----->*
**CENTER*
**JUSTIFY*
**TOP*
**BOTTOM*

*ITALIC ----- *YES ----->*
**NO*

BOLD ----- **YES* -----|
**NO*

Keywords

OBJECT This keyword is used to specify whether you want to define the width of a column view or a column cell format.
The possible values are:
**COLUMNVIEW* - if you want to define the width of a specified column view.
**CELLFORMAT* - if you want to define the format of the cells.
Both of these options has a different set of keywords available for it.
This keyword is mandatory.

COLUMN This keyword is used to specify the column number or range of column numbers you wish your **DEFINE** command to act on. You can specify a single column number or a comma-separated range.
The format of this keyword is as follows:
COLUMN(column), for example, **COLUMN**(3).
COLUMN(start column, end column), for example, **COLUMN**(3,5).
For the **COLUMNVIEW* option, the action acts on the whole column, while for the **CELLFORMAT* option you can also specify a specific row or rows to act on. See the **RANGE** keyword below for more details.
This keyword is mandatory.

If the **OBJECT** keyword is being set to **COLUMNVIEW* then the following keywords are available.

CLEAR This keyword is used to clear all defined column views.
This keyword is always used by itself. It cannot be used in conjunction with any other keywords.
The possible values are:

*YES - to clear all defined column views.
*NO - to not clear all defined column views.
The default value is *NO.

WIDTH This keyword is used to define the column width.
You must specify a numeric value here.
This keyword is mandatory.

If the OBJECT keyword is being set to *CELLFORMAT then the following keywords are available.

CLEAR This keyword is used to clear all defined cell formats.
This keyword is always used by itself. It cannot be used in conjunction with any other keywords.
The possible values are:
*YES - to clear all defined column views.
*NO - to not clear all defined column views.
The default value is *NO.

RANGE This keyword is used to define the range of rows within the specified column on which this DEFINE will act.
If left blank, then the entire column will be acted upon.
The format of this keyword is as follows:
RANGE(start row, end row), for example, RANGE(3,5).
This keyword is optional.

TYPE This keyword is used to define the column type.
The possible values are:
*NUMBER
*DATE
*BOOLEAN
*STRING
*BLANK

The default value is *NUMBER

This keyword is optional.

FORMAT

This keyword is used to define the cell format.

If the cell is of type *NUMBER, then the possible values you may use are as follows. Any valid NumberFormat may be used for these (i.e. #.##):

*ACCOUNTINGFLOAT ("\$,##0;(\$,##0)")

*ACCOUNTINGINTEGER ("\$,##0;(\$,##0)")

*ACCOUNTINGREDFLOAT ("\$,##0;[Red](\$,##0)")

*ACCOUNTINGREDINTEGER ("\$,##0;[Red](\$,##0)")

*DEFAULT ("#")

*EXPONENTIAL ("0.00E00")

*FLOAT ("0.00")

*FORMAT1 ("#,##0;(#,##0)")

*FORMAT2 ("#,##0;[Red](#,##0)")

*FORMAT3 ("#,##0.00;(#,##0.00)")

*FORMAT4 ("#,##0.00;[Red](#,##0.00)")

*FORMAT5 ("#,##0;(#,##0)")

*FORMAT6 ("#,##0;[Red](#,##0)")

*FORMAT7 ("#,##0.00;(#,##0.00)")

*FORMAT8 ("#,##0.00;[Red](#,##0.00)")

*FORMAT9 ("#,##0.00;(#,##0.00)")

*FORMAT10 ("##0.0E0")

*FRACTIONONEDIGIT ("?/?")

*FRACTIONTWO DIGITS ("??/??")

*INTEGER ("0")

*PERCENTFLOAT ("0.00%")

*PERCENTINTEGER ("0%")

*TEXT ("@")

*THOUSANDSFLOAT ("#,##0.00")

*THOUSANDSINTEGER ("#,##0")

The default value is *DEFAULT

If the cell is of type *DATE, then the possible values you may use are as follows. Any valid DateFormat may be used for these (i.e. dd mm yyyy):

*DEFAULT ("m/d/yy")

*FORMAT1 ("m/d/yy")

*FORMAT2 ("d-mmm-yy")

*FORMAT3 ("d-mmm")

*FORMAT4 ("mmm-yy ")

*FORMAT5 ("h:mm AM")

*FORMAT6 ("h:mm:ss AM")

*FORMAT7 ("H:mm")

*FORMAT8 ("H:mm:ss")

*FORMAT9 ("m/d/yy H:mm")

*FORMAT10 ("mm:ss")

*FORMAT11 ("H:mm:ss")

*FORMAT12 ("H:mm:ss.0")

The default value is *DEFAULT

This keyword is optional.

COMPLEXFORMAT This keyword is used to define the cell format of a type *NUMBER cell.

No validation checks are done on the format value, so the format must be a valid Excel number format. If it is not a valid format corruption to the Excel document can occur.

FONT

This keyword is used to define the font of the cell value.

The possible values are as follows:

*ARIAL

*COURIER

*TAHOMA

*TIMES

The JXL default font is Arial.

This keyword is optional.

FONTCOLOR

This keyword is used to define the font color.

The possible values are as follows:

*AQUA, *AUTOMATIC, *BLACK, *BLUE,
*BLUEGRAY, *BLUE2, *BRIGHTGREEN,
*BROWN, *CORAL, *DARKBLUE,
*DARKBLUE2, *DARKGREEN, *DARKPURPLE,
*DARKRED, *DARKRED2, *DARKTEAL,
*DARKYELLOW, *DEFAULTBACKGROUND,
*DEFAULTBACKGROUND1, *GOLD, *GRAY25,
*GRAY50, *GRAY80, *GREEN, *GRAY25,
*GRAY50, *GRAY80, *ICEBLUE, *INDIGO,
*IVORY, *LAVENDER, *LIGHTBLUE,
*LIGHTGREEN, *LIGHTORANGE,
*LIGHTTURQUOISE, *LIGHTTURQUOISE2,
*LIME, *OCEANBLUE, *OLIVEGREEN,
*ORANGE, *PALEBLUE, *PALETTEBLACK,
*PERIWINKLE, *PINK, *PINK2, *PLUM,
*PLUM2, *RED, *ROSE, *SEAGREEN,
*SKYBLUE, *TAN, *TEAL, *TEAL2,
*TURQUOISE, *TURQUOISE2,
*VERYLIGHTYELLOW, *VIOLET, *VIOLET2,
*WHITE, *YELLOW, *YELLOW2

The default value is *BLACK

This keyword is optional.

FONTSIZE

The keyword is used to define the font size.

The JXL default point size is 10.

This keyword is optional.

BORDER

This keyword is used to define the lines of the border.

The possible values are as follows:

*ALL

*NONE
*TOP
*BOTTOM
*LEFT
*RIGHT

There is no default value.

BORDERSTYLE

This keyword is used to define the borderstyle.

The BORDER keyword needs to be specified in conjunction with this keyword.

Possible values for this keyword are as follows:

*NONE
*MEDIUM
*THICK
*THIN
*HAIR
*DOUBLE

The default value is *THIN

This keyword is optional.

BORDERCOLOR

This keyword is used to define the border color.

The BORDER keyword needs to be specified in conjunction with this keyword.

The possible values are as follows:

*AQUA, *AUTOMATIC, *BLACK, *BLUE,
*BLUEGRAY, *BLUE2, *BRIGHTGREEN,
*BROWN, *CORAL, *DARKBLUE,
*DARKBLUE2, *DARKGREEN, *DARKPURPLE,
*DARKRED, *DARKRED2, *DARKTEAL,
*DARKYELLOW, *DEFAULTBACKGROUND,
*DEFAULTBACKGROUND1, *GOLD, *GRAY25,
*GRAY50, *GRAY80, *GREEN, *GRAY25,
*GRAY50, *GRAY80, *ICEBLUE, *INDIGO,
*IVORY, *LAVENDER, *LIGHTBLUE,
*LIGHTGREEN, *LIGHTORANGE,

*LIGHTTURQUOISE, *LIGHTTURQUOISE2,
*LIME, *OCEANBLUE, *OLIVEGREEN,
*ORANGE, *PALEBLUE, *PALETTEBLACK,
*PERIWINKLE, *PINK, *PINK2, *PLUM,
*PLUM2, *RED, *ROSE, *SEAGREEN,
*SKYBLUE, *TAN, *TEAL, *TEAL2,
*TURQUOISE, *TURQUOISE2,
*VERYLIGHTYELLOW, *VIOLET, *VIOLET2,
*WHITE, *YELLOW, *YELLOW2

The default value is *BLACK

This keyword is optional.

BACKGROUND

The possible values are as follows:

*AQUA, *AUTOMATIC, *BLACK, *BLUE,
*BLUEGRAY, *BLUE2, *BRIGHTGREEN,
*BROWN, *CORAL, *DARKBLUE,
*DARKBLUE2, *DARKGREEN, *DARKPURPLE,
*DARKRED, *DARKRED2, *DARKTEAL,
*DARKYELLOW, *DEFAULTBACKGROUND,
*DEFAULTBACKGROUND1, *GOLD, *GRAY25,
*GRAY50, *GRAY80, *GREEN, *GRAY25,
*GRAY50, *GRAY80, *ICEBLUE, *INDIGO,
*IVORY, *LAVENDER, *LIGHTBLUE,
*LIGHTGREEN, *LIGHTORANGE,
*LIGHTTURQUOISE, *LIGHTTURQUOISE2,
*LIME, *OCEANBLUE, *OLIVEGREEN,
*ORANGE, *PALEBLUE, *PALETTEBLACK,
*PERIWINKLE, *PINK, *PINK2, *PLUM,
*PLUM2, *RED, *ROSE, *SEAGREEN,
*SKYBLUE, *TAN, *TEAL, *TEAL2,
*TURQUOISE, *TURQUOISE2,
*VERYLIGHTYELLOW, *VIOLET, *VIOLET2,
*WHITE, *YELLOW, *YELLOW2

There is no default value.

This keyword is optional.

HALIGN

This keyword is used to define the horizontal alignment.

The possible values for this keyword are as follows:

*CENTRE

*CENTER

*FILL

*GENERAL

*JUSTIFY

*LEFT

*RIGHT

There is no default value.

This keyword is optional.

VALIGN

This keyword is used to define the vertical alignment.

The possible values for this keyword are as follows:

*CENTRE

*CENTER

*JUSTIFY

*TOP

*BOTTOM

There is no default value.

This keyword is optional.

ITALIC

This keyword is used to set the font to italics or not.

The possible values for this keyword are:

*NO

*YES

The default value is *NO

This keyword is optional.

BOLD

This keyword is used to set the font to bold or not.

The possible values for this keyword are:

*NO

*YES

The default value is *NO

This keyword is optional.

CLOSE

The CLOSE command closes any workbooks currently open for reading or writing. When the workbooks are closed all the associated internal objects are released and are available for garbage collection.

```
CLOSE ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(CLOSE)  
TO_GET(#JSMSTS #JSMMSG)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 close) to_get(#jsmxsts  
#jsmxmsg)
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values. When used in this way, there is a possibility of conflict between application-defined keyword names and those used by LANSAs Integrator. To reduce the possibility of conflict, your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

```
SERVICE_GET ----- PROPERTY ----- value -----  
>  
  
    >-- TRACE ----- *SERVICE-----|  
                *TRANSPORT
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

TRACE *SERVICE - read service trace file.
*TRANSPORT - read transport trace file.

Examples

The examples retrieve the value of the com.acme.property.messageType (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html" will be returned to the application in this case):

com.acme.property.messageType=html

RDML

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'

RDMLX

#jsmcmd := ' service_get property(com.acme.property.messageType)'
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD)  
TO_GET(#JSMSTS #JSMMSG)
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhde1 service_unload)  
to_get(#jsmxsts #jsmxmsg)
```

5.37.4 ExcelReadService Examples

Examples are supplied for:

[RDML](#)

[RDMLX](#)

5.37.5 Troubleshooting

The FILE keyword is still supported on the READ and WRITE commands for compatibility with earlier version of LANSAs Integrator, however it is recommended that you use the following structure for any new development:

- To READ from a spreadsheet (the OPEN keyword MODE is *READ by default):

```
OPEN FILE(ABC.XLS)
```

```
READ
```

```
CLOSE
```

- To WRITE to a spreadsheet:

```
OPEN FILE(ABC.XLS) MODE(*WRITE)
```

```
WRITE
```

```
CLOSE
```

5.38 SQLService

The SQLService is designed to enable users to develop applications that can communicate with any database that can be accessed with a JDBC driver. While the SQLService could be run in a number of different scenarios, the most common usage of it would be to enable IBM i applications (written in LANSAs, RPG, Cobol or any other IBM i language) to access data on other platforms via a JDBC driver.

Some examples of its use might be:

- A IBM i (LANSAs, RPG, Cobol) application accessing data in an SQL Server database on Windows server.
- A IBM i (LANSAs, RPG, Cobol) application accessing data in an Oracle database on a Linux server.
- A IBM i (LANSAs, RPG, Cobol) application accessing data in a DB2 database residing on another IBM i machine.

The above describes IBM i applications accessing data on other types of databases on other machines, but this is not a restriction. You may have a requirement that a Windows based application needs to talk to a JDBC driver to access IBM i data for example.

Using this service with a JDBC driver will provide you with a simpler way of reading and updating records in other databases. It also enables real-time updates and access of data on remote servers. Other techniques to achieve the same objective might include the use of dataqueues or staging files which store data until a polling program on another platform comes along to pick up the data and process it into the other database. Such processes are fraught with risk as they consume CPU and network resources, even when information is not being processed. Refer to [5.38.1 What can I use the SQLService for?](#) for further information of how you might use the service.

Related Services

There are no related services.

Technical Specifications

When using the SQLService, a key requirement is to obtain a JDBC driver for the database that you want to connect to. You will have to obtain this yourself from the database vendor. Most vendors will provide the requisite drivers on their installation CDs, as well as allow you to download them from their web site. In many cases these drivers will be available at no charge. Most databases

will need you to use a driver that is of exactly the same version as the database. Therefore, ensure that you have access to the correct driver for the version and type of database that you wish to connect to. Vendor web sites are the best source for this information. Some useful sites are as follows:

Oracle JDBC Drivers:

www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html

Microsoft SQL Server Drivers:

www.microsoft.com/technet/downloads/sqlsrvr.msp

Adaptive Server Drivers:

www.sybase.com/products/informationmanagement/softwaredeveloperkit/jconnect

NOTE: For many databases, drivers can also be obtained from third party sources.

The IBM Toolbox for Java comes with two JDBC drivers for the IBM i, which can be used for accessing DB2 data:

- IBM Toolbox for Java JDBC Driver - this is a Type 4 driver, that makes direct socket connect to the database host server
- IBM Developer Kit for Java JDBC Driver - this is a Type 2 driver, and makes native method calls to the SQL CLI (Client Level Interface).

You would use one of these if you have a requirement to access DB2 data on an IBM i via a JDBC driver. In general, you would use the native driver if your program is intended to only run on an IBM i JVM and the data is on the same machine. You would tend to use the Toolbox driver if your program is intended to run on other JVMs or the Java program is on one IBM i OS system and the data in on a separate IBM i OS system. For more information on the IBM Toolbox for Java please refer to [IBM Toolbox for Java](#).

This service assumes a basic knowledge of SQL. The SET command provides the ability to control numerous commitment control features for your SQL statements. While these are addressed in some detail in this document, a complete explanation of the effect of each type is beyond the scope of this document.

Finally, ensure that your remote server is available on your network and that you have all the connections set up so that you can access the database.

5.38.1 What can I use the SQLService for?

The most common use of this service will be for an IBM i application to access data on another server - whether it is a DB2 database on another IBM i, an Oracle database on a Linux box, an SQL Server on a Windows machine, or any other permutation where data can be accessed via a JDBC driver. The data access required may be create, read, update, and or delete.

There are a number of ways to achieve the same objective in terms of accessing remote databases, but JDBC can often be the simplest, hence the SQLService is an option you should consider when assessing how to tackle such issues.

The following are some examples that demonstrate where this service might be used.

Example 1

There are many examples of web applications needing to access information from multiple servers. You may have a LANSAs for the Web application running on an IBM i server. The application accesses most of its information from the IBM i, but you may have some screens that need to present data that is gathered from the IBM i plus some data that is residing on an Oracle database on a Windows server. For the data residing on the IBM i, you would FETCH, SELECT, INSERT, UPDATE or DELETE this information in the usual manner you would as for any LANSAs IBM i application. For the data residing in the Oracle database though, you should seriously consider using the SQLService to access this database via a JDBC driver.

Now there are other ways that this goal can be achieved using LANSAs. You may decide to place your data into dataqueue or a staging file on the IBM i, and have a Visual LANSAs application polling this dataqueue or staging file then transferring the data to and from the DB2 and Oracle databases as required. For simple solutions, this can be quite effective. But it does have some drawbacks:

- The approach uses the concept of pulling the data from the IBM i. In this case, the web application itself is reliant on another system to do the updating.
- If the polling of the Visual LANSAs application is not regular, then the Web application will possibly suffer from asynchronous and less than real-time data. This problem may be overcome by increasing the polling rate, but of course this would consume more CPU and network resources. Even if no data needs to be update, the polling will still occur to interrogate the data queue or staging file - because this is the only way it can find out if data

needs to be moved in one direction or the other.

By using the SQLService with a JDBC driver, you can ensure that data is accessed and updated to and from the web user as and when required. This is because the database access is handled by your LANSAs for the Web WebEvent function or WAM itself, and not relying on a polling application to transfer data. So, with this approach, you use CPU and network resources only when needed.

Example 2

You may ask, "Why would an IBM i application need to use a JDBC driver to access IBM i DB2 data?" (as provided for by the two drivers that come with the IBM Toolbox for Java).

Such an approach might be very useful when you want to access a DB2 database that is residing on another remote IBM i. An organisation might, for example, have a number of IBM i machines in different locations around the world. From time to time a LANSAs, RPG, or Cobol application running in one country may need to access data from the DB2 database residing on the IBM i sitting in another country. This is a very good example of where the SQLService could be used. The SQLService could be used to access the remote DB2 database real-time via the IBM Toolbox for Java JDBC Driver. This driver comes with all IBM i installations, so it makes for a simple cost effective mechanism for IBM i applications to access remote DB2 databases.

5.38.2 Using the SQLService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, an application that needs to access a remote database would typically issue the following sequence of commands:

JSM(X)_OPENJSM(X)_COMMANDs

SERVICE_LOAD

CONNECT

SET

EXECUTE

READ

COMMIT

ROLLBACK

METADATA

DISCONNECT

SERVICE_UNLOAD

JSM(X)_CLOSE

Refer to the [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

5.38.3 SQLService Properties

The following lists the contents of the SQLService.properties as they are shipped in a standard LANSA.

The first two sections are the most important.

The first section is the JDBC driver details (for example, driver.db2=com.ibm.as400.AS400JDBCdriver). The driver ID that you specify in the CONNECT command will search for a related driver in the SQLService.properties file (on the left-hand side of the '=' sign). The name of the actual driver to use appears on the right hand side of the '=' sign. This JDBC driver must reside in the jar directory of your LANSA Integrator instance. It is your responsibility to ensure that it is there.

The second section specifies the database details (for example, database.info=jdbc:as400://DBHOST/LIBRARY;naming=sql;errors=format=iso;translate binary=true). The database that you specify in the CONNECT command will search for a related database in the SQLService.properties file (on the left hand side of the '=' sign). The name and location of the actual database to use is indicated on the right hand side of the '=' sign. You will note that the database detail may also include some start up details.

```
#!<studio-project id="20000000-000000" name="lansa">
#
# SQLService resource ( Default )
#
driver.db2=com.ibm.as400.access.AS400JDBCdriver
driver.oracle=com.ddtek.jdbc.oracle.OracleDriver
driver.sqlserver=com.ddtek.jdbc.sqlserver.SQLServerDriver
# driver.sqlserver=com.microsoft.jdbc.sqlserver.SQLServerDriver
driver.sqlanywhere=com.sybase.jdbc2.jdbc.SybDriver
#
# database.info=jdbc:microsoft:sqlserver://DBHOST
# database.northwind=jdbc:microsoft:sqlserver://10.10.10.10:1433;databasena
# database.info=jdbc:datadirect:sqlserver://DBHOST:1433
# database.info=jdbc:datadirect:oracle://DBHOST:1521;sid=oemrep
# database.info=jdbc:sybase:Tds:DBHOST:2638?ServiceName=dbname
database.info=jdbc:as400://DBHOST/LIBRARY;naming=sql;errors=full;date f
```


message.001=Service successfully loaded
message.002=Service successfully unloaded
message.003=Command is not supported :
message.004=Missing DATABASE keyword
message.005=Missing DATABASE resource from service properties
message.006=Missing DRIVER keyword
message.007=Missing DRIVER resource from service properties
message.008=No current database connection
message.009=EXECUTE command requires QUERY, UPDATE, PREPARED
message.010=Require list to receive result
message.011=No SQL command
message.012=No result set available for reading
message.013=Column count does not match list field count
message.014=Field mapped column does not exist in result set :
message.015=Result set is empty
message.016=Require parameter map list
message.017=Field column map requires a two field list
message.018=Map field does not exist in function :
message.019=Missing TABLE keyword
message.020=Require working list
message.021=Working list requires four fields
message.022=No suitable driver found
message.023=No CALL command
message.024=Unsupported call type :
message.025=Cannot connect to database

#!</studio-project>

5.38.4 SQLService Commands

Your application issues commands to the SQLService by passing the command strings through the Java Service Manager using the JSM_COMMAND or the JSMX_COMMAND Built In Function, or an API for your chosen development language.

The commands that the SQLService processes are:

SERVICE_LOAD

CONNECT

SET

GET

EXECUTE

READ

COMMIT

ROLLBACK

METADATA

DISCONNECT

SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
--->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR
```

```
>-- TRACE_NAME ----- name -----|  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER
```

For more information refer to:

[Service Program Tracing from the Client](#)

Keywords

- | | |
|---------|--|
| SERVICE | The name of the service to be loaded - in this case SQLService. |
| TRACE | To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command.
The possible values for the TRACE keyword are: <ul style="list-style-type: none">• *NO• *YES• *ERROR The trace option *ERROR will turn on tracing and if the |

service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.

TRACE_NAME This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory.

Special keyword values are also available for the TRACE_NAME keyword.

- *SERVICE
- *PROCESS
- *FUNCTION
- *JOBNAME
- *JOBUSER
- *JOBNUMBER

Examples

RDML

* Define the field to hold the JSM command

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(SQLService  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
#jsmcommand := 'service_load service(SQLService)'
```

```
use builtin(jsmx_command) with_args(#jsmxhdl1 #jsmcommand) to_get(#jsn
```

CONNECT

The CONNECT command is used to connect to the remote database. You must do this before you can perform any activity on the database.

```
CONNECT ----- DRIVER ----- value -----  
>  
  
  >-- DATABASE ----- value ----->  
  
  >-- USER ----- value ----->  
  
  >-- PASSWORD ----- value -----|
```

Keywords

- DRIVER** This keyword is used to specify the name of the JDBC driver you intend to use to access the remote database.
The value specified here must correspond to a key entry in the SQLService properties file. This reference will provide the details of the full name and path of the JDBC driver.
This keyword is mandatory
- DATABASE** This keyword is used to specify the database that you wish to connect to with the JDBC driver.
The value specified here must correspond to a key entry in the SQLService properties file. This reference will provide the details of the full name and path of the database.
This keyword is mandatory.
- USER** This keyword is used to hold the User ID that you intend to use to connect to the database.
The keyword is mandatory.
- PASSWORD** This keyword is used to hold the password for the User ID that you intend to use to connect to the database.
The keyword is mandatory.

Comments / Warnings

By default the connection is made in auto-connect mode, which means it automatically commits changes after each statement.

Any additional database connection properties will need to be placed into the SQLService properties file.

Examples

RDML

* Define the JSM message fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

```
CHANGE FIELD(#JSMCMD) TO('CONNECT DRIVER(DB2) DATABASE(  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

* Define the JSM Command related fields

```
Define Field(#JSMSTS) Type(*Char) Length(020)
```

```
Define Field(#JSMMSG) Type(*Char) Length(256)
```

```
Define Field(#JSMCMD) Type(*Char) Length(256)
```

```
Define Field(#JSMHND) Type(*Char) Length(4)
```

```
#JSMCMD := 'Connect Driver(DB2) Database(JSMJDBC) User(SMITH) Pass  
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

SET

The SET command is used to set database connection properties before you actually perform any activity on the database. It is very important that you understand the full capabilities of this command.

```
SET ----- READONLY ----- *YES -----
>
      *NO

>-- MAXROWS ----- n ----->
      *NOMAX

>-- AUTOCOMMIT ----- *YES ----->
      *NO

>-- ISOLATION ----- *NONE ----->
      *READCOMMIT
      *READUNCOMMITTED
      *REPEATABLEREAD
      *SERIALIZABLE

>-- PARAMETER ----- *LIST ----->
      *SQL
      *CALL
      *MAP
      *NONE

>-- ONERROR ----- *DISCONNECT -----
->
      *CONTINUE
      *ROLLBACK

>-- ONWARNING ----- *STOP ----->
      *CONTINUE

>-- SQLSTATE ----- *NONE -----|
      *ERROR
```

**WARNING*

**ALL*

Keywords

- READONLY** This keyword indicates if the current database connection should allow read or write access. A database connection must first be established using the `CONNECT` command. This keyword then applies to the current database connection.
- By default when a database connection is established, read and write access is permitted (depending on restrictions imposed by the database).
- A value of **YES* indicates any changes to the database are automatically committed after executing each statement.
- A value of **NO* indicates that changes to the database are controlled by programmed transaction boundaries.
- Please refer to Comments / Warnings for this command for further information.
- MAXROWS** This keyword specifies the maximum number of rows to be returned by a query.
- AUTOCOMMIT** This keyword indicates if information should be automatically committed to the database. A database connection must first be established using the `CONNECT` command to use this keyword. This keyword then applies to the current database connection.
- By default when a connection is established it is in auto-commit mode.
- A value of **YES* indicates any changes to the database are automatically committed after executing each statement.
- A value of **NO* indicates that changes to the database are controlled by programmed transaction boundaries.
- Please refer to Comments / Warnings for this command for further information.
- ISOLATION** This keyword indicates the level of transaction isolation to

be applied to the current database. A database connection must be established using the CONNECT command to use this keyword. This keyword then applies to the current database connection.

The default value is *NONE otherwise you can choose from these four levels of transaction defined by the ANSI/ISO SQL standard:

***READUNCOMMITTED**

All uncommitted data is readable from any connection. This is the same as not having any isolation (*NONE).

***READCOMMITTED**

This prevents dirty reads but does not prevent phantoms or non-repeatable reads. Using this isolation level, only data committed before the current transaction began will be available. Any dirty data or changes made by concurrent transactions will not be available.

This level is obviously more restrictive than the *READUNCOMMITTED.

***REPEATABLEREAD**

This prevents dirty and non-repeatable reads but does not prevent phantom rows. This means the probability of other transactions having to wait for this one are increased when compared to *READUNCOMMITTED and *READCOMMITTED

This is more restrictive than *READCOMMITTED.

***SERIALIZABLE**

*SERIALIZABLE provides the highest transaction isolation. When a transaction is isolated at the *SERIALIZABLE level, only data committed before the transaction began is available. Neither dirty data nor concurrent transaction changes committed during transaction execution are available. This level emulates serial transaction execution, as transactions will effectively be executed one after another rather than concurrently.

This is more restrictive than *REPEATABLEREAD.

In relation to these isolation levels there are three

phenomena that you need to understand before you can determine the correct isolation level to apply to your application, namely:

- Dirty Reads - A transaction reads data written by an uncommitted transaction. If the second transaction is rolled back, the data read by the first transaction is then invalid because the rollback undoes the changes. The first transaction won't be aware that the data it has read has become invalid.
- Non-repeatable Reads - A transaction re-reads data it has previously read and finds that data has been modified by another committed transaction.
- Phantom Read - Phantom reads occur when new records added to the database are detectable by transactions that started prior to the insert. A transaction re-executes a query and returns a set of rows satisfying a search condition only to find that additional rows satisfying the condition have been inserted by another committed transaction.

The ANSI/ISO SQL standard isolation levels and the corresponding behaviors are summarized in a table in Comments / Warnings following.

PARAMETER This keyword can be used to prepare the SQL statement prior to running the EXECUTE or READ commands. The value you specify here depends very much on the type of activity you wish to perform on the database.

There are four possible values for this keyword:

- *LIST - you may set this value if you intend to use the UPDATE keyword of the EXECUTE command. When taking this approach the values that you intend to add to, update, or delete from the database will be specified in a list that is included in the SERVICE_LIST of this command. It is recommended that you use this option if you are going to be updating (or inserting or deleting) more than a single record. (otherwise you would have to run the EXECUTE as many times as there are records, and this would place unnecessary load on your system

and network). Please refer to the following Lists and Variables and the Examples for this command.

- *SQL - you may set this value if to intend to prepare the SQL statement in advance of the EXECUTE command. By doing so, you place your SQL statement into a working list that is passed in the SERVICE_LIST of this command. This feature is particularly useful if you are going to be using long SQL statements. This is particularly useful for RDML applications where the longest field length can only be 256 characters. Please refer to the following Lists and Variables and Examples for this command for further information.
- *CALL - you may set this value if you intend to use call procedures that have IN, OUT and INOUT parameters.
- *MAP - if you are expecting a large result set to be returned from a query, then you should choose to use the READ command to receive the list of values. To use the READ command you need to set this keyword to *MAP to indicate that the SERVICE_LIST of this command will contain the column and field mapping information. Please refer to the following Lists and Variables and the Examples for this command for further information.
- *NONE - you may use this to clear the current map, list and SQL statements.

ONERROR

The keyword is used to indicate the action you want to take if an exception is encountered during the execution of the SQL statement.

There are three possible options:

- *ROLLBACK - if an exception is encountered during the execution of the SQL statement, a rollback will be initiated.
- *CONTINUE - if an exception is encountered during the execution of the SQL statement, the processing will continue.
- *DISCONNECT - if an exception is encountered during the execution of the SQL statement, the database connection will be closed.

The default value is *DISCONNECT.

ONWARNING This keyword is used to indicate the action you wish to take if a warning message is issued during the execution of the SQL statement.

There are two possible values:

- *CONTINUE - if a warning is encountered during the execution of the SQL statement, the processing will continue.
- *STOP - - if a warning is encountered during the execution of the SQL statement, the command will be stopped.

The default value is *STOP.

SQLSTATE By default, if an SQLException occurs then the JSMSTS field is set to ERROR, and if an SQLWarning occurs then the JSMSTS field is set to WARNING.

This keyword may be used to allow the actual SQLxxxx code to be returned in place of these default values.

There are four possible values:

- *ALL - this will ensure that an SQLException or an SQLWarning returns the SQLxxxx status.
- *ERROR - this will ensure that an SQLException returns the SQLxxxx status while an SQLWarning returns a WARNING status.
- *WARNING ERROR - this will ensure that an SQLWarning returns the SQLxxxx status while an SQLException returns an ERROR status.
- *NONE - this will ensure that an SQLException returns ERROR status and an SQLWarning returns WARNING status.

The default value is *NONE.

Comments / Warnings

JDBC drivers have vendor defined default settings for such areas as commitment control and exception handling. As an example, autocommit is normally switched on.

The ANSI/ISO SQL standard isolation levels and the corresponding behaviors are summarized in the following table.

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read
*READUNCOMMITTED	Possible	Possible	Possible
*READCOMMITTED	Not possible	Possible	Possible
*REPEATABLEREAD	Not possible	Not possible	Possible
*SERIALIZABLE	Not possible	Not possible	Not possible

The SQLService SET command provides you with the capability to override these settings to meet your own requirements.

A detailed explanation of commitment control and error handling is beyond the scope of this documentation. If you are unfamiliar with isolation levels, commitment control and error handling then it is strongly recommended that you research these topics through other channels. These topics are quite generic so academic books or web sites might be a good place to start. You are also recommended to review any material that the database vendor provides on these topics.

It is strongly recommended that you test your commit and rollback logic extensively.

Lists and Variables

The PARAMETER keyword provides some very useful techniques for preparing your SQL statements. These can make you code simpler and in many cases help you produce more efficient applications.

If you choose to use the PARAMETER keyword with a value of *LIST, *SQL, *MAP or *CALL, then you will need to supply a working list with the SET command. The information that the working list contains will depend upon which of the three values you specify.

The following provides some information on what each value is used for as well

as the list information that needs to be passed with the command:

***LIST** - this value will allow you to pass a list of records to be inserted, updated, or deleted from the remote database. This list will be sent to the remote database with a prepared SQL statement and the prepared SQL statement will be run once for each record in the list. One of the benefits of this feature is a reduction in network and system resources. The alternative would be to run the SQL statement inside a loop and send it to the server for every separate record that needs to be updated. The working list provided with this value must provide one column for each field required by the SQL statement. The order of the columns is important as is explained in the EXECUTE command.

***SQL** - this value will allow you to prepare an SQL statement prior to the EXECUTE command. This is particularly useful when your SQL statement is very large (for example, the longest field type for RDML applications is 256 characters, so when you use this in your JSM_COMMAND the available space to include your SQL statement is even less than this). When you set this value your large SQL statement may be placed into a working list. If need be, the statement may be placed over many rows of the working list. (Note: only one statement per SET command). The working list passed with this option must contain a single character field, which will contain the SQL statement.

***CALL** this value will allow you to pass a list of IN, OUT and INOUT stored procedure call parameters. The working list argument needs to have three columns. These columns are Type, Direction and Value. Possible values for the Type column are *CHAR, *VARCHAR, *SMALLINT, *INTEGER, *FLOAT, *DOUBLE, *DECIMAL and *NUMERIC. Possible values for the Direction column are *IN, *OUT and *INOUT. The Value column is a string value.

***MAP** - if you are expecting to receive a large result set from your query of the database, then you should opt to use the READ command to obtain the values in the result set. When you take this approach, a working list will need to be passed with this SET command that contains the column field mappings. The *MAP value indicates that your list is intended for this purpose. The working list that you pass will contain two columns that will describe the column/field mappings. That is, the first column will contain the name of the field used in this function to hold the value returned from the remote table, and the second column will hold the name of the related field as it is named in the remote table. There will be as many rows in this list as

there are fields to be returned from the query. Refer to the READ command for further information on this feature and where to use it.

Please note that it is quite possible that you could use a number of these options in conjunction with each other. Refer to the following examples and the SQLService Examples to see how this works. For information on how to supply a working list to service commands from RDML or RDMLX applications, refer to [5.38.5 SQLService Examples](#).

Examples

The SET, EXECUTE, and READ commands of the SQLService service are very tightly related to each other. As such, you are recommended to review the more extensive examples in [SQLService Examples](#).

RDML

* Define the field to hold the JSM command

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

Define the field and list that will hold the SQL command

```
DEFINE FIELD(#COLCMD) TYPE(*CHAR) LENGTH(100)
```

```
DEF_LIST NAME(#WRKCMD) FIELDS(#COLCMD) TYPE(*WORKING)
```

* SET the commitment control settings

```
CHANGE FIELD(#JSMCMD) TO('SET ISOLATION(*READCOMMITTED)  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

* Set up the SQL statement in the list then run the SET command

```
CHANGE FIELD(#COLCMD) TO('SELECT ID,NAME,AGE,SALARY')
```

```
ADD_ENTRY TO_LIST(#WRKCMD)
```

```
CHANGE FIELD(#COLCMD) TO('FROM TBLNAME')
```

```
ADD_ENTRY TO_LIST(#WRKCMD)
```

```
CHANGE FIELD(#JSMCMD) TO('SET PARAMETER(*SQL) SERVICE_LI  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

* Define the JSM Command related fields

```
Define Field(#JSMSTS) Type(*Char) Length(020)
```

```
Define Field(#JSMMSG) Type(*Char) Length(256)
```

```
Define Field(#JSMCMD) Type(*Char) Length(256)
Define Field(#JSMHND) Type(*Char) Length(4)
```

* Define the field and the list to hold an SQL statement

```
Define Field(#COLCMD) Type(*Char) Length(020)
Def_List Name(#WRKCMD) Fields(#COLCMD) Type(*WORKING)
```

* Set up the commitment control settings

```
#JSMCMD := 'Set Isolation(*READCOMMITTED) AutoCommit(*NO) OnE1
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

* Build an SQL statement then run the Set command

```
#COLCMD := 'Select ID,NAME,AGE,SALARY'
Add_Entry To_List(#WRKCMD)
#COLCMD := 'From TBLNAME'
Add_Entry To_List(#WRKCMD)
#JSMCMD := 'Set Parameter(*SQL) Service_List(COLCMD)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

GET

The GET command is used to get return values from a call to a stored procedure or to get the row number that is in error from a parameter list in a prepared statement.

```
GET ----- OBJECT ----- *PARAMETERLISTROW ----  
-----|  
                                *PARAMETERCALL  
                                *NEXTRESULT
```

Keywords

OBJECT The *PARAMETERLISTROW is used to return the list entry that has caused the error.

It will return the first erroneous row number found into the JSM message field of the JSM command Built-In Function.

The *PARAMETERCALL is used to return the stored procedure call parameters

The *NEXTRESULT is used to moved to the next result set. If not result set is available then status code NORESULT is returned.

Comments / Warnings

This command is very useful when a prepared statement with a parameter list has returned an exception, such as a duplicate key error. Using this command will allow you to ascertain which row in the list is causing the error, so that you may then take some corrective measures.

The typical flow of using this command might be as follows:

Step 1: Execute a prepared statement.

Step 2: If the returned status is OK, then continue with your processing.

Step 3: If the returned status is not OK, then use the GET command to establish which row is in error.

Examples

RDML

* Define the JSM command and message fields

```

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
*
* Define the field to hold the INSERT statement
DEFINE FIELD(#COLCMD) TYPE(*CHAR) LENGTH(100)
DEF_LIST NAME(#WRKCMD) FIELDS(#COLCMD) TYPE(*WORKING)
*
* Define the fields used in the working list
DEFINE FIELD(#COL1) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#COL2) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#COL3) TYPE(*DEC) LENGTH(8) DECIMALS(0)
DEFINE FIELD(#COL4) TYPE(*DEC) LENGTH(12) DECIMALS(2)
* Define the working list to hold the values
* to be used by the INSERT statement
DEF_LIST NAME(#WRKLST) FIELDS(#COL1 #COL2 #COL3 #COL4) TY
*
* Create bind values
CHANGE FIELD(#COL1) TO(B2001)
CHANGE FIELD(#COL2) TO('Tom')
CHANGE FIELD(#COL3) TO(45)
CHANGE FIELD(#COL4) TO(35000.60)
ADD_ENTRY TO_LIST(#WRKLST)
* (*****
CHANGE FIELD(#COL1) TO(A2012)
CHANGE FIELD(#COL2) TO('Antony')
CHANGE FIELD(#COL3) TO(45)
CHANGE FIELD(#COL4) TO(35000.60)
ADD_ENTRY TO_LIST(#WRKLST)
*
* Prepare the INSERT
CHANGE FIELD(#COLCMD) TO('INSERT INTO TBLNAME(ID,NAME,A
ADD_ENTRY TO_LIST(#WRKCMD)
CHANGE FIELD(#JSMCMD) TO('SET PARAMETER(*SQL) SERVICE_LI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
*
* Prepare the list to contain the data to be inserted
CHANGE FIELD(#JSMCMD) TO('SET PARAMETER(*LIST) SERVICE_LI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS

```

*

* Execute the prepared statement

```
CHANGE FIELD(#JSMCMD) TO('EXECUTE PREPARED(*SQLPARAMET  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

*

* If an error occur, find out the problem row

```
IF COND('#JSMSTS *NE OK')
```

```
CHANGE FIELD(#JSMCMD) TO('GET OBJECT(*PARAMETERLISTROW  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

*

```
DISPLAY FIELDS(#JSMMSG)
```

```
ENDIF
```

RDMLX

* Define the JSM command and message fields

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
```

```
Define Field(#JSMMSG) Type(*CHAR) Length(256)
```

```
Define Field(#JSMCMD) Type(*CHAR) Length(256)
```

```
Define Field(#JSMHND) Type(*Char) Length(4)
```

*

* Define the field to hold the INSERT statement

```
Define Field(#COLCMD) Type(*CHAR) Length(100)
```

```
Def_List Name(#WRKCMD) Fields(#COLCMD) Type(*WORKING)
```

*

* Define the fields and list that will contains the result set returned from the qu

```
Define Field(#COL1) Type(*CHAR) Length(010)
```

```
Define Field(#COL2) Type(*CHAR) Length(020)
```

```
Define Field(#COL3) Type(*DEC) Length(008) Decimals(0)
```

```
Define Field(#COL4) Type(*DEC) Length(012) Decimals(0)
```

```
Def_List Name(#WRKLST) Fields(#COL1 #COL2 #COL3) Type(*WORKIN
```

*

* Create bind values

```
#COL1 := 'B2001'
```

```
#COL2 := 'Tom'
```

```
#COL3 := 45
```

```
#COL4 := 35000.60
```

```
Add_Entry To_list(#Wrklst)
```

```

*
#COL1 := 'B2002'
#COL2 := 'Jones'
#COL3 := 23
#COL4 := 22000.60
Add_Entry To_list(#Wrklst)
*
* Prepare the INSERT
#COLCMD := 'Insert Into TBLNAME(ID,NAME,AGE,SALARY) VALUES(
Add_Entry To_list(#Wrkcmd)
#JSMCMD := 'Set Parameter(*SQL) Service_List(COLCMD)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
*
* Prepare the list to contain the data to be inserted
#JSMCMD := 'Set Parameter(*LIST) Service_List(COL1,COL2,COL3,COL4)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
*
* Execute the prepared statement
#JSMCMD := 'Execute Prepared(*SQLPARAMETER)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
*
If '#JSMSTS *ne Ok'
* If an error occur, find the problem row - Get command will return the row nu
#JSMCMD := 'Get Object(*PARAMETERLISTROW)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
ENDIF

```

EXECUTE

The EXECUTE command is used to run SQL statement against the remote database.

Conditional

```
EXECUTE ----- QUERY ----- value -----  
>  
    *SQLPARAMETER  
  
>-- UPDATE ----- value ----->  
    *SQLPARAMETER  
  
>-- PREPARED ----- value ----->  
    *SQLPARAMETER  
  
>-- CALL ----- value ----->  
  
>-- CALLTYPE ----- *QUERY ----->  
    *UPDATE  
    *EXECUTE  
  
>-- CALLSYNTAX ----- *JDBC ----->  
    *ORACLE  
  
>-- RETURN ----- *NONE -----|  
    *CHAR  
    *STRING  
    *SMALLINT  
    *INTEGER  
    *FLOAT  
    *DOUBLE  
    *DECIMAL  
    *NUMERIC  
    *ORACLECURSOR
```

Keywords

QUERY

You will use this keyword to execute a query on the database where you expect to have a result set returned from your SQL statement.

The keyword may be specified in one of two ways:

- You may specify the SQL statement directly into this keyword. For example, `SELECT 'value','value' FROM table`
- You will use the `*SQLPARAMETER` value if you have prepared an SQL statement using the `PARAMETER(*SQL)` keyword on the SET command.

See the Comments / Warnings for further information on this.

UPDATE

You will use this command if you wish to issue an SQL update and are not expecting a result set to be returned.

The keyword may be specified in one of two ways:

- You may specify the SQL statement directly into this keyword. For example, `UPDATE table SET col1='value'`, or `DELETE FROM table WHERE col1='value'`.
- You will use the `*SQLPARAMETER` value if you have prepared an SQL statement using the `PARAMETER(*SQL)` keyword on the SET command.

See the Comments / Warnings for further information on this.

PREPARED

This keyword allows you to prepare an SQL statement. In most case you will need to make data available to the prepared SQL statement, and in such cases you will need to prepare these values in a working list using the `SET PARAMETER(*LIST)` command.

The keyword may be specified in one of two ways:

- You may specify the SQL statement directly into this keyword. For example, `UPDATE table SET col1=?,` or `DELETE FROM table WHERE col1=?`. Note the presence of the question marks - this is explained in Comments / Warnings below.

You will use the `*SQLPARAMETER` value if you have

prepared an SQL statement using the PARAMETER(*SQL) keyword on the SET command.

See Comments / Warnings for further information on this.

CALL

You will use this keyword to call a stored procedure to run your SQL statement. This stored procedure will be residing on the remote server.

See the Comments / Warnings for further information on this.

CALLTYPE

This keyword is used to specify the type of SQL statement that the stored procedure specified in the CALL command is.

There are three possible values for this keyword:

- *QUERY - if the SQL statement in the stored procedure is a query, then the CALL will return the result set into a working list.
- *UPDATE - if the SQL statement in the stored procedure is an update, insert, or delete then the CALL will return a row count (of the number of records updated) into a working list.
- *EXECUTE - if this keyword is specified then the stored procedure will return nothing.

The default value is *QUERY.

Refer to the following Lists and Variables for further comments on the lists returned when using this keyword.

This keyword must be used in conjunction with the CALL keyword only.

CALLSYNTAX This keyword is used to specify whether you are using JDBC syntax or Oracle JDBC syntax.

There are two possible values for this keyword:

- *JDBC - use this to indicate that you are using standard JDBC syntax.
- *ORACLE - use this to indicate that you are using Oracle JDBC syntax.

Please refer to the following Comments / Warnings for

further details on the use of this keyword.

This keyword must be used in conjunction with the CALL keyword only.

RETURN

This keyword is used to specify the datatype of the return value from a stored procedure / function.

The following values are possible

- *NONE - indicates that no value is being returned
- *CHAR - indicates that a value of datatype CHAR is being returned
- *STRING - indicates that a value of datatype STRING is being returned
- *SMALLINT - indicates that a value of datatype SMALLINT is being returned
- *INTEGER - indicates that a value of datatype INTEGER is being returned
- *FLOAT - indicates that a value of datatype FLOAT is being returned
- *DOUBLE - indicates that a value of datatype DOUBLE is being returned
- *DECIMAL - indicates that a value of datatype DECIMAL is being returned
- *NUMERIC - indicates that a value of datatype NUMERIC is being returned
- *ORACLECURSOR - indicates that the value being returned is a cursor that in turn will be used to access the result set from an Oracle query. This can only be used on an Oracle database with the Oracle driver

The default value is *NONE.

Please refer to the following Comments / Warnings for further details on the use of this keyword.

This keyword must be used in conjunction with the CALL keyword only.

Comments / Warnings

The EXECUTE command is very powerful. It is important that you understand

its capability in order to take full advantage of it.

The following details when you should use each command and how best to use it. It is recommended that you read this thoroughly before you proceed.

QUERY

If you want to run a query on a remote database and you are expecting a result set to be returned (for example, values to be returned from a select), then you would typically use the EXECUTE QUERY(SQL statement) command. Refer to the following List and Variables for information on how to retrieve the result set.

If you want to use a very large SQL statement then you have the option of preparing the statement using the SET PARAMETER(*SQL) command, then using the EXECUTE QUERY(*SQLPARAMETER) command.

Also refer to the CALL and PREPARED keywords.

UPDATE

The EXECUTE UPDATE(SQL statement) command will typically be used when you want to update the database using an insert, update, or delete. A result set will not be returned. With this in mind, you will therefore not use this to query data.

If you want to use a very large SQL statement then you have the option of preparing the statement using the SET PARAMETER(*SQL) command, then using the EXECUTE QUERY(*SQLPARAMETER) command.

Also refer to the CALL and PREPARED keywords.

PREPARED

If you were intending to run the same SQL statement more than once then you would be recommended to use the PREPARED command, as it will normally reduce the execution time.

This is achieved by sending the prepared statement to the database and compiling it once. This means that each time it is executed it can run immediately without having to be compiled first. Without taking this approach, we would have to put our UPDATE command into a loop and the statement would have to re-establish a connection to the database every single time.

The prepared SQL statement can be run with no parameters, but in most cases the advantage is gained for those that take parameters. Parameters are passed to the SQL statement by a working list created in conjunction with the SET PARAMETER(*LIST) command. So, for example, you might want to update

the address details for a number of employees in one go. Using a prepared SQL and passing a list of the data to be updated will be the most efficient way of doing it.

If you want to use a very large SQL statement then you have the option of preparing the statement using the SET PARAMETER(*SQL) command, then using the EXECUTE QUERY(*SQLPARAMETER) command.

If a prepared SQL statement starts with INSERT and ends with VALUES(*CALC) or VALUES(*ALL) then special parsing of the SQL statement is done. The *CALC option means that a binding parameter for column is calculated and the statement is modified. The *ALL option means that a binding parameter for each column is calculated and all other columns in the table are included using a default value.

CALL

This keyword allows you to call a stored procedure on the remote server. This stored procedure will contain your SQL statement. The CALL command must be used in conjunction with the CALLTYPE command to specify whether it is of type *QUERY, *UPDATE, or *EXECUTE.

Stored procedures can provide a very efficient way of running your SQL statements. For example, they can reduce network traffic if a lot of data is being returned. You are encouraged to research this topic in more detail. The IBM Infocentre and Redbooks, both available from the IBM web site, have a wealth of information on this topic.

Parameters can be passed to the remote procedure using a working list and the SET PARAMETER(*LIST) command. Refer to the following Lists and Variables for information on how to do this.

Some stored procedure calls return a warning to say that they been executed, so the warning check needs to be turned off using the SET ONWARNING(*CONTINUE) command. The warning error code 466 (result sets are available from procedure) is ignored and is not treated as a warning.

Some examples of this command and how to create stored procedures are provided in the SQLService Examples. Some examples of stored procedures can be found at the end of this section.

CALLSYNTAX

The Oracle JDBC Driver does not fully support the JDBC syntax when it is used to return a value from a database function. It uses its own Oracle syntax. With this in mind, you need to use this keyword to indicate whether or not you

are using standard JDBC (in which case specify *JDBC) or the Oracle JDBC (in which case specify *ORACLE). This will tell the SQLService what style of JDBC driver it should be prepared for.

RETURN

Stored procedures are able to specify a return value. This is something different to returning a result set. A result set is a list of one or more records or values returned from a query. Whereas a return value is a single value that a stored procedure returns upon completion. Typical stored procedure syntax will have a "RETURN" or similar command (according to the target database syntax) and associated field name as one of the last commands executed by the stored procedure. This value could be anything, but some common examples might be a field containing the number of records selected or updated, or the maximum value, or some flag.

This RETURN keyword allows you to indicate the datatype for the returned value. The datatypes possible will depend upon what your target database and JDBC driver can support. The IBM i, as an example, only currently support a datatype of integer for values returned from a stored procedure. It is ultimately your responsibility to be aware of the datatypes that your target database and JDBC driver can handle.

Please refer to the Notes on Oracle if you are using this SQLService with an Oracle database.

Notes on Oracle

The Oracle database does not fully support the JDBC syntax when used to return a value from a database function. It will only work with Oracle syntax. This is why you need to specify the CALLSYNTAX as *ORACLE, so that the SQLService knows what to expect and how to handle it.

Oracle has two types of callable programs. One is a stored procedure and the other is a function. A stored procedure cannot return a value, so you must use a function if you want to return one.

The Oracle JDBC driver and the Oracle database do not support the Java JDBC "ResultSet resultSet = call.executeQuery ()" method. For Oracle to return a result set to a JDBC client, an Oracle function needs to be created that returns an Oracle cursor and the JDBC call.execute () method needs to be used. If your target database is indeed Oracle and you are expecting a result set to be returned from the stored procedure, then you must specify *ORACLECURSOR as the RETURN value for this EXECUTE command. This will then be used by the SQLService to determine which records to return. This explicit requirement to

state that a cursor is being returned is only relevant to Oracle. Other databases do not need to use this keyword in association with retrieving the result set. When using this value in your keyword your Oracle function will need to have a "RETURN cursorname" command so that the cursor is returned to the SQLService.

Further to this, the JDBC client needs to prepare the call using Oracle syntax and not the industry standard JDBC syntax.

An example of an Oracle function is provided in Examples following.

Handling Parameters Using the SET PARAMETER(*LIST) Command

If you are running an EXECUTE command in conjunction with a SET PARAMETER(*LIST) command then a list of the data is being passed to the remote database along with the SQL statement. When taking this approach your SQL statement will need to include a '?' for each value that needs to be inserted into the command. These are referred to as parameter binding positions.

Essentially, the first column in the list will be placed where the first '?' is, the second column will be placed where the second '?' is and so forth. The code examples provide some good examples of how this works.

Notes on SQL Syntax

Please also note that the syntax that you use will be relevant to the your target database, so you will need to know the syntax for the JDBC driver that you are using.

You will need to place quotes around string values inside your SQL statements. For example:

```
EXECUTE UPDATE(UPDATE TABLE SET COL1='value')  
EXECUTE UPDATE(DELETE FROM TABLE WHERE COL1='value')
```

If your SQL statement contains open or closed brackets, you will need to include double quotes around the entire statement. For example:

```
EXECUTE PREPARED("INSERT INTO TABLE (COL1,COL2,COL3)
```

Lists and Variables

If you are using one of the following - the QUERY keyword, the PREPARED keyword where the SQL statement is a SELECT, or a CALL keyword with a CALLTYPE of *QUERY, then you may supply a working list with this command for the result set to be returned back to.

This working list will contain the fields that you are expecting to be returned from the query. The list will be filled in field sequence order. The select

statement may actually return more fields than are indicated in this list, but there cannot be more list fields than columns returned. The column value is received from the result set using the `resultSet.getString (column index)` method. The list entry field is set with this string value and Java data type to native data type conversion is done.

If you are expecting very large lists to be returned then you may omit this working list and access the result set using the `READ` command.

The `UPDATE` keyword will not return a result set so a working list need not be supplied in this situation.

A `CALL` of `CALLTYPE *UPDATE` will return a row count (number of records updated) so a one column list may be supplied to capture this value.

If you need to pass parameters to a called procedure (using the `CALL` command), you will need to supply a working list with the parameters in it. This working list will have as many columns as parameters you need to pass. Only the first row of the working list will be used. The list will need to be prepared using the `SET PARAMETER(*LIST)` command.

Refer to the following examples and the [SQLService Examples](#) to see how this works. For information on how to supply a working list to service commands from `RDML`, `RDMLX` or `3GL` applications, refer to [Java Service Manager Clients](#).

Examples

The `SET`, `EXECUTE`, and `READ` commands of the `SQLService` service are very tightly related to each other. As such, you are recommended to review the more extensive examples in [SQLService Examples](#).

RDML

* Define the JSM command and message fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Define the fields and list that will contains the result set returned from the qu

```
DEFINE FIELD(#COL1) TYPE(*CHAR) LENGTH(010)
```

```
DEFINE FIELD(#COL2) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#COL3) TYPE(*DEC) LENGTH(008) DECIMALS(0)
```

```
DEF_LIST NAME(#WRKLST) FIELDS(#COL1 #COL2 #COL3) TYPE(*W(
```

* Define the field and list that will be passed to the remote procedure to indicate
DEFINE FIELD(#PARAM1) TYPE(*CHAR) LENGTH(010)
DEF_LIST NAME(#PARAMLIST) FIELDS(#PARAM1) TYPE(*WORKING)
CHANGE FIELD(#PARAM1) TO(A1001)
ADD_ENTRY TO_LIST(#PARAMLIST)

* Set up the list so that it is passed to the remote procedure when the EXECUTE
CHANGE FIELD(#JSMCMD) TO('SET PARAMETER(*LIST) SERVICE_LIST
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSM

* Call procedure
CHANGE FIELD(#JSMCMD) TO('EXECUTE CALL("CALLSELECT(?)")')
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSM

RDMLX

* Define the JSM command and message fields
Define Field(#JSMSTS) Type(*CHAR) Length(020)
Define Field(#JSMMSG) Type(*CHAR) Length(256)
Define Field(#JSMCMD) Type(*CHAR) Length(256)
Define Field(#JSMHND) Type(*Char) Length(4)

* Define the fields and list that will contain the result set returned from the query
Define Field(#COL1) Type(*CHAR) Length(010)
Define Field(#COL2) Type(*CHAR) Length(020)
Define Field(#COL3) Type(*DEC) Length(008) Decimals(0)
Def_List Name(#WRKLIST) FIELDS(#COL1 #COL2 #COL3) Type(*WORKING)

* Define the field and list that will be passed to the remote procedure to indicate
Define Field(#PARAM1) Type(*CHAR) Length(010)
Def_List Name(#PARAMLIST) FIELDS(#PARAM1) Type(*WORKING)
#PARAM1 := A1001
Add_Entry To_List(#PARAMLIST)

* Set up the list so that it is passed to the remote procedure when the EXECUTE
#JSMCMD := 'Set Parameter(*LIST) Service_List(PARAM1)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get

* Call procedure

```
#JSMCMD := 'Execute("CallSelect(?)") CallType(*QUERY) Service_List(CO  
Use Builtin (JSM_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get(;
```

IBM i Examples of Stored Procedures:

To create a stored procedure for an IBM i database you need to edit a source member and use a member type of TXT.

The following are some examples of three IBM i stored procedures. You will notice that the first line specifies the name of the stored procedure. So, for example, the first one is called CALLSELECT and its location is the JSMJDBC library. The second and third examples demonstrate how a value is returned.

Detailed instructions on creating stored procedures is beyond the scope of this documentation. You are therefore encouraged to research this topic in more detail. A good place to start is the IBM Infocentre and Redbooks, both on the IBM web site as they have a wealth of information on this topic.

```
***** Beginning of data *****
```

```
CREATE PROCEDURE JSMJDBC/CALLSELECT(IN CODE CHAR (10))  
LANGUAGE SQL  
READS SQL DATA  
RESULT SETS 1
```

```
BEGIN  
DECLARE SQLCODE INTEGER DEFAULT 0;  
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

```
DECLARE C1 CURSOR WITH RETURN TO CLIENT FOR
```

```
SELECT ID,NAME,AGE FROM JSMJDBC/TBLNAME WHERE ID = COD
```

```
OPEN C1;
```

```
END
```

```
***** End of data *****
```

```
***** Beginning of data *****
```

```
CREATE PROCEDURE JSMJDBC/CALLUPDATE(IN CODE CHAR (10))
```

```
LANGUAGE SQL
MODIFIES SQL DATA
```

```
BEGIN
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
DECLARE num_records INTEGER;
```

```
UPDATE JSMJDBC/TBLNAME SET SALARY=12000.43 WHERE ID = CO
```

```
GET DIAGNOSTICS num_records = ROW_COUNT;
```

```
RETURN num_records;
```

```
END
```

```
***** End of data *****
```

```
***** Beginning of data *****
```

```
CREATE PROCEDURE JSMJDBC/CALLEXECUT(IN CODE CHAR (10))
LANGUAGE SQL
MODIFIES SQL DATA
```

```
BEGIN
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
DECLARE num_flag INTEGER DEFAULT 0;
```

```
UPDATE JSMJDBC/TBLNAME SET SALARY=16000.26 WHERE ID = CO
```

```
RETURN num_flag;
```

```
END
```

```
***** End of data *****
```

To create the stored procedures you then need to use the RUNSQLSTM command. For example:

```
RUNSQLSTM SRCFILE(JSMJDBC/QCLSRC) SRCMBR(CALLSELECT) C
```

When it comes to running these stored procedures with the SQLService you can use something like the following EXECUTE command

```
EXECUTE CALL("CALLSELECT(?)") CALLTYPE(*QUERY)
```

```
EXECUTE CALL("CALLUPDATE(?)") CALLTYPE(*UPDATE) RETURN(*)
```

```
EXECUTE CALL("CALLEXECUT(?)") CALLTYPE(*EXECUTE) RETURN(*)
```

Oracle Example of a Database Function:

The following is an example of an Oracle database function that is creating a result set. As such, note that it is returning a cursor.

Detailed instructions on creating stored procedures is beyond the scope of this documentation. You are therefore encouraged to research this topic in more detail via other channels.

```
CREATE OR REPLACE FUNCTION "SYSTEM"."TEST" ( PARAM1 IN CH
BEGIN
OPEN CURSOR_1 FOR SELECT * FROM MYTABLE WHERE FLD_1='AI
RETURN CURSOR_1 ;
END;
```

When it comes to running this function with the SQLService you can use something like the following EXECUTE command

```
EXECUTE CALL("TEST(?,?)") CALLTYPE(*EXECUTE) CALLSYNTAX(*)
```

READ

The READ command is used to read a result set returned by the SQL statement specified in the EXECUTE command. It uses the optional COLUMN_LIST keyword values or the field and column mapping supplied by the SET PARAMETER(*MAP) command.

If the EXECUTE command does not have a working list to receive the result list into, then it will remain on the server until accessed by a READ command.

You should use the READ command when the number of expected records will exceed the size or width of a working list. The READ command enables you to reduce the number of records brought down from the server at any one time. If the SQL result is going to return 20,000 records, for example, it would be better to read them into sets of 1,000, reducing the amount of resources being allocated.

As another example, the maximum width of an RDML working list is 256 characters. If you are expecting a result set of 300 characters in width then you could not return this into a working list as a part of the EXECUTE command. In this case you would use the READ command to retrieve the records in manageable chunks. If need be, you could issue multiple READs and place them results into lists of less than 256 characters in width.

```
READ ----- SCROLL ----- *YES ----->
                          *NO
>-- COLUMN_LIST ----- value -----|
```

Keywords

SCROLL

This keyword is used to specify whether new rows or the old rows are selected for the next read.

There are two options:

- *YES - this will fill the list argument with data then scroll forward
- *NO - this will fill the list argument with data then not scroll forward

Refer to Comments / Warnings for further information on this.

The default value is *YES.

COLUMN_LIST A comma separated list of columns that exist in the result set.

The sequence of the columns must match the fields in the working list.

Comments / Warnings

The READ command reads the current result set using the field and column mapping supplied by the SET PARAMETER (*MAP) command. Working list fields are mapped to columns and data for these columns are mapped to the working list fields.

As mentioned above the power of this READ command is its ability to enable you to access information in more manageable chunks. The SCROLL keyword enables you to decide whether your subsequent READ will be retrieving new columns for the same records as this current read, or columns from the next set of records. That is, if you select:

- *YES - it will select the columns specified in the READ, then scroll forward so that you next read will be retrieving columns from a completely new set of records
- *NO - it will select the columns in the READ, and not scroll forward so that your next READ may select new columns from the same rows.

The number of records it scrolls forward will be that number defined as the number of entries in your working list.

Lists and Variables

This command will need to supply a working list to which the retrieved values are returned. The columns defined in the working list will be those that you are wish to retrieve for this particular READ. The relationship between these fields and the actual fields in the remote table will have been set up in the SET PARAMETER(*MAP) command.

Refer to the following examples and the SQLService Examples to see how this works. For information on how to supply a working list to service commands from RDML, RDMLX or 3GL applications, refer to [Java Service Manager Clients](#).

Examples

The SET, EXECUTE, and READ commands of the SQLService service are very tightly related to each other. As such, you are recommended to review the

more extensive examples in the [SQLService Examples](#).

RDML

* Define the JSM command and message fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

* Define the fields and the list that will be used to indicate the field/column mapping

```
DEFINE FIELD(#FIELD) TYPE(*CHAR) LENGTH(010)
DEFINE FIELD(#COLUMN) TYPE(*CHAR) LENGTH(030)
DEF_LIST NAME(#MAPLST) FIELDS(#FIELD #COLUMN) TYPE(*WORD)
```

* Define the fields and the lists that the queried data will be returned back into

```
DEFINE FIELD(#COL1) TYPE(*CHAR) LENGTH(010)
DEFINE FIELD(#COL2) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#COL3) TYPE(*DEC) LENGTH(008) DECIMALS(0)
DEFINE FIELD(#COL4) TYPE(*DEC) LENGTH(012) DECIMALS(2)
DEF_LIST NAME(#WRKLST1) FIELDS(#COL1 #COL3) TYPE(*WORKING)
DEF_LIST NAME(#WRKLST2) FIELDS(#COL1 #COL2 #COL4) TYPE(*WORKING)
```

* Define the mapping

```
CHANGE FIELD(#FIELD) TO(COL1)
CHANGE FIELD(#COLUMN) TO(ID)
ADD_ENTRY TO_LIST(#MAPLST)
CHANGE FIELD(#FIELD) TO(COL2)
CHANGE FIELD(#COLUMN) TO(NAME)
ADD_ENTRY TO_LIST(#MAPLST)
CHANGE FIELD(#FIELD) TO(COL3)
CHANGE FIELD(#COLUMN) TO(AGE)
ADD_ENTRY TO_LIST(#MAPLST)
CHANGE FIELD(#FIELD) TO(COL4)
CHANGE FIELD(#COLUMN) TO(SALARY)
ADD_ENTRY TO_LIST(#MAPLST)
```

```
CHANGE FIELD(#JSMCMD) TO('SET PARAMETER(*MAP) SERVICE_LIST')
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMMSG)
```

CHANGE FIELD(#JSMCMD) TO('EXECUTE QUERY(SELECT ID,NAME,
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS

USE BUILTIN(JSM_COMMAND) WITH_ARGS('READ SERVICE_LIST(C

USE BUILTIN(JSM_COMMAND) WITH_ARGS('READ SERVICE_LIST(C

RDMLX

* Define the JSM command and message fields

Define Field(#JSMSTS) Type(*CHAR) Length(020)

Define Field(#JSMMSG) Type(*CHAR) Length(256)

Define Field(#JSMCMD) Type(*CHAR) Length(256)

Define Field(#JSMHND) Type(*Char) Length(4)

* Define the fields and the list that will be used to indicate the field/column ma

Define Field(#FIELD) Type(*Char) Length(010)

Define Field(#COLUMN) Type(*Char) Length(030)

Def_List Name(#MAPLST) Fields(#FIELD #COLUMN) Type(*Working)

* Define the fields and the lists that the queried data will be returned back into

Define Field(#COL1) Type(*Char) Length(010)

Define Field(#COL2) Type(*Char) Length(020)

Define Field(#COL3) Type(*Dec) Length(008) Decimals(0)

Define Field(#COL4) Type(*Dec) Length(012) Decimals(2)

Def_List Name(#WRKLST1) Fields(#COL1 #COL3) Type(*Working)

Def_List Name(#WRKLST2) Fields(#COL1 #COL2 #COL4) Type(*Working)

* Define the mapping

#FIELD := COL1

#COLUMN := ID

Add_Entry To_List(#MAPLST)

#FIELD := COL2

#COLUMN := NAME

Add_Entry To_List(#MAPLST)

#FIELD := COL3

#COLUMN := AGE

Add_Entry To_List(#MAPLST)

```
#FIELD := COL4
#COLUMN := SALARY
Add_Entry To_List(#MAPLST)
```

```
#JSMCMD := 'Set Parameter(*Map) Service_List(FIELD,COLUMN)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

```
* Run the Query
```

```
#JSMCMD := 'Execute Query(Select ID,NAME,AGE,SALARY From TBLN/'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

```
* Read the columns 1 & 3 (which hold ID and AGE)
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND 'Read Service_List((
```

```
* Read the columns 1, 2, & 4 (which hold ID, NAME, and SALARY)
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND 'Read Service_List((
```

COMMIT

The COMMIT command is used to commit an SQL transaction.

```
COMMIT ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Comments / Warnings

It is your responsibility to ensure that commitment control has been started and configured if you are using an IBM i. If it is not, then an error will occur.

Examples

RDML

* Define the JSM message fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(COMMIT) TO_GET(#JSM
```

RDMLX

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
```

```
Define Field(#JSMMSG) Type(*CHAR) Length(256)
```

```
Define Field(#JSMHND) Type(*Char) Length(4)
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND Commit) To_Get(#J
```

ROLLBACK

The ROLLBACK command is used to rollback an SQL transaction.

```
ROLLBACK ----- no keywords -----|
```

Keywords

There are no keywords associated with this command.

Comments / Warnings

It is your responsibility to ensure that commitment control has been started and configured if you are using an IBM i. If it is not, then an error will occur.

Examples

RDML

* Define the JSM message fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(ROLLBACK) TO_GET(#J
```

RDMLX

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
```

```
Define Field(#JSMMSG) Type(*CHAR) Length(256)
```

```
Define Field(#JSMHND) Type(*Char) Length(4)
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND RollBack) To_Get(#
```

METADATA

The METADATA command is used to access the database table column name, column type, column precision and column scale information.

```
METADATA ----- TABLE ----- value -----|
```

Keywords

TABLE This keyword is used to indicate the table you want to retrieve the metadata for.

This keyword is mandatory.

Lists and Variables

This command requires a working list to return the table metadata back to. This working list should be made available in the TO_GET portion of this JSM_COMMAND. The working list will require 4 fields:

- The first field will hold the column name.
- The second field will hold the column type.
- The third field will hold the column precision.
- The fourth field will hold the column scale.

You may size the fields to fit the information you are expecting back from the remote database. Different databases may have different lengths and types for this information.

Examples

RDML

* Define the JSM command and message fields

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(256)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(256)
```

Define the fields and list that meta data will be returned back into

```
DEFINE FIELD(#COLNAME) TYPE(*CHAR) LENGTH(030)
```

```
DEFINE FIELD(#COLTYPE) TYPE(*CHAR) LENGTH(030)
```

```
DEFINE FIELD(#COLPRCSN) TYPE(*DEC) LENGTH(003) DECIMALS(0)
```

```
DEFINE FIELD(#COLSCALE) TYPE(*DEC) LENGTH(003) DECIMALS(0)
```

```
DEF_LIST NAME(#WRKLST) FIELDS(#COLNAME #COLTYPE #COLPR
```

Prepare and execute the command

```
CHANGE FIELD(#JSMCMD) TO('METADATA TABLE(TBLNAME) SERV  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
```

```
Define Field(#JSMMSG) Type(*CHAR) Length(256)
```

```
Define Field(#JSMCMD) Type(*CHAR) Length(256)
```

```
Define Field(#JSMHND) Type(*Char) Length(4)
```

Define the fields and list that meta data will be returned back into

```
Define Field(#COLNAME) Type(*CHAR) Length(030)
```

```
Define Field(#COLTYPE) Type(*CHAR) Length(030)
```

```
Define Field(#COLPRCSN) Type(*DEC) Length(003) DECIMALS(0)
```

```
Define Field(#COLSCALE) Type(*DEC) Length(003) DECIMALS(0)
```

```
Def_List Name(#WRKLST) Fields(#COLNAME #COLTYPE #COLPRCSN #
```

Prepare and execute the command

```
#JSMCMD := 'Metadata Table(TBLNAME) Service_List(COLNAME,COLTY  
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

DISCONNECT

The DISCONNECT keyword is used to disconnect from the current connected database.

DISCONNECT ----- no keywords -----|

Keywords

This command has no keywords.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('DISCONNECT') TO_GET
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdl1 'disconnect') to_get(#jsmxs
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service and removes the temporary directory.

```
SERVICE_UNLOAD ----- no keywords -----  
|
```

Keywords

There are no keywords associated with this command.

Examples

RDML

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO
```

RDMLX

```
use builtin(jsmx_command) with_args(#jsmxhdle1 service_unload) to_get(#jsi
```

5.38.5 SQLService Examples

To help clarify the use of the SQLService in RDML, particularly the different options in the SET PARAMETER (value), the EXECUTE and the READ commands, three RDML examples have been provided.

[RDML](#)

[RDMLX](#)

5.39 BASE64FileService

Service Name: **BASE64FileService**

The BASE64FileService allows files to be encoded to BASE64 format and decoded from BASE64 format.

The BASE64FileService supports the following commands:

5.39.1 SERVICE_LOAD

5.39.2 SERVICE_GET

5.39.3 ENCODE

5.39.4 DECODE

5.39.5 DELETE

5.39.6 RENAME

5.39.7 SET

5.39.8 LIST

5.39.9 SERVICE_UNLOAD

5.39.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD			
--------------	--	--	--

5.39.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.39.3 ENCODE

The ENCODE command will BASE64 encode the contents of specified FROM file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
ENCODE	FROM	value	Mandatory. File Path.
	TO	value	Mandatory. File Path.
	FOLD	*YES	Optional. Add CRLF every 76 characters.
		*NO	Default.
	REPLACE	*YES	Optional. Replace existing file.
		*NO	Default.

5.39.4 DECODE

The DECODE command will BASE64 decode the contents of the specified FROM file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
DECODE	FROM	value	Mandatory. File Path.
	TO	value	Mandatory. File Path.
	REPLACE	*YES	Optional. Replace existing file.
		*NO	Default.

5.39.5 DELETE

The DELETE command deletes the specified file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
DELETE	FILE	value	Mandatory. File Path.

5.39.6 RENAME

The RENAME command renames the specified local file to a new name. The TO file must be in the same directory as the original, FROM file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RENAME	FROM	value	Mandatory. Old file name.
	TO	value	Mandatory. New file name.
	REPLACE	*YES	Optional. Replace file.
		*NO	Do not replace file. The default value of the REPLACE keyword is *NO. If the TO filename exists, then the rename process is stopped, unless the REPLACE keyword is *YES.

5.39.7 SET

The SET command sets the current working directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	DIR	value	Mandatory. Directory.

5.39.8 LIST

The LIST command will fill the command's list object with absolute file names. A single field working list is required to receive the canonical file paths.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value	
LIST	DIR	value	Optional. Fill working list with directory listing. All files in that directory will be selected.	
		EXT	value	Optional. Filtering on file extension. The filtering match is case insensitive.
		SORT	*NONE	Optional. Allows sorting of file names. Default. No sorting
	*NAME		Sort on file name.	
	*MODIFIED		Sort on modified date.	
	REVERSE	*YES	Optional. Reverse the order of the sort.	
*NO		Default.		

If a directory path specified by the DIR keyword is used then filenames in that directory will be selected.

Example

```
LIST DIR(/document) EXT(PDF) SERVICE_LIST(PATH)
```

This will return a list of filenames in the /document directory that have a PDF extension.

5.39.9 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.40 PGPSFileService

Service Name: **PGPSFileService**

The PGPSFileService allows files to be encoded to PGP format and decoded from PGP format.

Use any OpenPGP tool to create and manage for keys.

<https://www.gnupg.org>

The GNU gpg2 command-line tool allows you to create and manage keys on your PC.

<http://www.gpg4win.org>

The gpg2 tool maintains the public and secret key collection files in the following folder.

C:\Users\XXX\AppData\Roaming\gnupg\pubring.gpg

C:\Users\XXX\AppData\Roaming\gnupg\secring.gpg

These collection files, pubring.gpg and secring.gpg need to be copied to the LI instance pki/ folder.

The PGPSFileService will use these files for locating recipient public keys for encryption and the secret key for decryption.

The PGPSFileService supports the following commands:

5.40.1 SERVICE_LOAD

5.40.2 SERVICE_GET

5.40.3 ENCODE

5.40.4 DECODE

5.40.5 DELETE

5.40.6 RENAME

5.40.7 SET

5.40.8 LIST

5.40.9 SERVICE_UNLOAD

5.40.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD			
--------------	--	--	--

5.40.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.40.3 ENCODE

The ENCODE command will PGP encode the contents of specified FROM file. When encrypting a single column working list is required to supply the list of recipients.

The recipient name can be either a partial userID or the keyID of the recipient's public key.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
ENCODE	FROM	value	Mandatory. File Path.
	TO	value	Mandatory. File Path.
	REPLACE	*YES	Optional. Replace existing file.
		*NO	Default.
	NAME	value	Optional. Literal data name.
		*TOFILE	Use TO file name.
		*FROMFILE	Use FROM file name.
		*CONSOLE	Default.
	CIPHER	*NONE	Optional. Encryption cipher.
		*DES	
		*3DES	
		*IDEA	
		*CAST5	Default.
		*SAFER	
		*AES-128	
		*AES-192	
		*AES-256	

*TWO FISH

*BLOW FISH

SIGNER	value	The signer value is comprised of two parts, the name and password in the format name:password. The name can be the keyID or the partial userID of the secret key.
--------	-------	---

5.40.4 DECODE

The DECODE command will PGP decode the contents of the specified FROM file.

The JSM response message could contain additional information about whether the a signature verification or integrity verification have failed.

The JSM response message could contain none or one of the following values.

SIGNATURE_FAILED

INTEGRITY_FAILED

SIGNATURE_VERIFIED

INTEGRITY_VERIFIED

SIGNATURE_FAILED,INTEGRITY_FAILED

SIGNATURE_FAILED,INTEGRITY_VERIFIED

SIGNATURE_VERIFIED,INTEGRITY_FAILED

SIGNATURE_VERIFIED,INTEGRITY_VERIFIED

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
DECODE	FROM	value	Mandatory. File Path.
	TO	value	Mandatory. File Path.
	REPLACE	*YES	Optional. Replace existing file.
		*NO	Default.
	PASSWORD	value	The password to extract the private key from the secret key. The secret key is automatically selected using the recipient public keys within the encrypted message.

5.40.5 DELETE

The DELETE command deletes the specified file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
DELETE	FILE	value	Mandatory. File Path.

5.40.6 RENAME

The RENAME command renames the specified local file to a new name. The TO file must be in the same directory as the original, FROM file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RENAME	FROM	value	Mandatory. Old file name.
	TO	value	Mandatory. New file name.
	REPLACE	*YES	Optional. Replace file.
		*NO	Do not replace file. The default value of the REPLACE keyword is *NO. If the TO filename exists, then the rename process is stopped, unless the REPLACE keyword is *YES.

5.40.7 SET

The SET command sets the current working directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SET	DIR	value	Mandatory. Directory.

5.40.8 LIST

The LIST command will fill the command's list object with absolute file names. A single field working list is required to receive the canonical file paths.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
LIST	DIR	value	Optional. Fill working list with directory listing. All files in that directory will be selected.
		value	Optional. Filtering on file extension. The filtering match is case insensitive.
	SORT	*NONE	Optional. Allows sorting of file names. Default. No sorting
		*NAME	Sort on file name.
		*MODIFIED	Sort on modified date.
	REVERSE	*YES	Optional. Reverse the order of the sort.
*NO		Default.	

If a directory path specified by the DIR keyword is used then filenames in that directory will be selected.

Example

```
LIST DIR(/document) EXT(PDF) SERVICE_LIST(PATH)
```

This will return a list of filenames in the /document directory that have a PDF extension.

5.40.9 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

5.41 AxesTerminalService

The AxesTerminalService is designed to permit a JSM client application to connect to and interact with an aXes Terminal Server to navigate, populate and interrogate IBM i 5250 screens. Using the AxesTerminalService, an application can interact directly with existing IBM i 5250 applications running on the same or a different server system. The application can issue commands to perform such operations as:

- connect (and logon) and disconnect the 5250 session
- interrogate the values of current screen identifiers and field values
- set the values of 5250 screen entry fields
- send function keys such as ENTER and F3
- perform multiple operations with a single command by using aXes terminal operation scripts

aXes is a separately licensed LANSAs product that Web enables 5250 applications by dynamically transforming the 5250 data stream into a graphical interface for display in a Web browser.

The AxesTerminalService interacts with aXes to allow your application to drive the 5250 application via the HTTP protocol.

In general, if the existing 5250 application with which you seek to integrate has documented, published application program interfaces that provide the interaction you require, then using those APIs is probably the better option for integrating with it.

However, where that is not the case, being able to drive the 5250 screens may provide an expedient means to implement a level of integration with the application that may not otherwise be possible.

Before implementing such a solution you should understand its limitations and the extent to which you are prepared to accept them in the pursuit of expediency. Any application that seeks to interact with another application via its 5250 screens (by any means) assumes risks - for example, the 5250 application's screens may change or certain inputs provided may yield results that were not anticipated. These risks may manifest themselves as unanticipated and unhandled application failures and/or cost and difficulty of maintaining the solution. Such manifestations are not the fault of the tools (in this case aXes or the AxesTerminalService), but, rather, are inherent in the nature of the solution.

Notes:

1. The aXes Terminal Server is a separately licensed product. Additional software license fees apply.
2. A correctly installed, licensed and configured aXes Terminal Server that is network-accessible to the JSM client application is a necessary pre-requisite for using the AxesTerminalService.
3. The aXes Terminal Server (and hence the 5250 applications) do not have to reside on the same computer system that is executing the AxesTerminalService. For example your application may execute on a Windows server and drive a 5250 application on a separate IBM i server.
4. The AxesTerminalService is intended primarily for use with LANSAR DML or RDMLX client functions and components. If your client is coded in a 3GL language on Windows or IBM i servers, aXes provides separate APIs for those languages that may be a more appropriate solution.

Technical Specifications

The AxesTerminalService works with aXes version 2.1 and above.

5.41.1 What can I use the AxesTerminalService for?

You can use the AxesTerminalService to allow your application to interact with any 5250 application, whether or not you have source code or specifications for the 5250 application. You only need to know how to operate the application as a 5250 terminal operator would.

The following paragraphs outline some example scenarios that might make use of the AxesTerminalService.

To receive and enter sales orders electronically ...

When your ERP application was developed and implemented, all sales orders were received by telephone operators who entered the details at 5250 screens.

Now, many of your customers wish to submit their sales orders electronically in XML format. Unfortunately, details of the internals of the ERP application for the order entry screens are not available.

In order to satisfy the customer requirement without replacing the ERP application, you can elect to implement a program that can read the received sales order XML files (perhaps using LANSAs Integrator's XMLBindService), extract the details and use the AxesTerminalService to drive the 5250 screens of the ERP order entry function.

To provide a sales order status enquiry as a web service ...

In addition to electronic submission of sales orders, your largest customer now demands that you implement a web service that enables them to electronically interrogate your system to determine the current status of their sales orders.

Currently, this information is only accessible via a 5250 screen enquiry and you have neither database specification nor application program interfaces to enable you to satisfy this requirement.

Instead, you implement the web service using LANSAs Integrator's SOAP Server wizard and the SOAPServerService. In your implementation code, you take advantage of the AxesTerminalService to drive the 5250 application such that your program can navigate to the order status screen for the specified sales order number and extract the required details.

5.41.2 Using the AxesTerminalService

Whether you are writing your client application in RDML / RDMLX or in a 3GL such as RPG, you have to complete the same basic steps.

For example, the simplest application that connects to an aXes Terminal Server to fill one screen value and send ("enter") would typically issue the following sequence of service commands:

JSM(X)_OPEN

```
JSM(X)_COMMANDs
  SERVICE_LOAD
  CONNECT
  SETBYNAME
  SEND
  DISCONNECT
  SERVICE_UNLOAD
```

JSM(X)_CLOSE

Refer to the [Java Service Manager Clients](#) for the command details that apply to your chosen development language.

Return variables

Note that the AxesTerminalService commands make extensive use of client program variables named in command keywords to return variable information to the client program. Such command keywords have names beginning with 'RTN'.

For example, to retrieve the value of a field on the current 5250 screen by reference to its screen row and column numbers and place the result in the client program variable named MYVALUE, a LANSAP RDMLX client application might contain code like this:

```
define field(#myvalue) reffld(#std_text)

#jsmcommand := 'getbypos row(10) col(44) rtnvalue(MYVALUE) SERVICE_

use builtin(jsmx_command) with_args(#jsmhandle #jsmcommand) to_get(#jsn

if (#myvalue = 'XYZ')

...
```

endif

In this example, the presence of `SERVICE_EXCHANGE(*FIELD)` in the JSM service command string is essential in order to exchange the program's variables with the service and permits the service to address the variables to return the requested values.

If you omit the `SERVICE_EXCHANGE(*FIELD)` from the JSM command, the service will not have access to the named variable and the program will not receive the value.

Notes:

1. Where necessary, `SERVICE_EXCHANGE(*FIELD)` must be specified in uppercase exactly as shown.
2. This is a simplified description. There are other means to accomplish the exchange of program variables between the client application and the JSM service.

5.41.3 AxesTerminalService Scripts

The AxesTerminalService provides service commands that permit your client application to interact with your 5250 application via aXes in virtually any way necessary.

However, most of the service commands perform a single operation amongst several or many that may be necessary to complete just one screen interaction. For example, the SETBYNAME or SETBYPOS service commands let you set the value in a 5250 screen entry field, but only a single field for each execution. This level of interface provides the most flexibility to your application. At each and every operation, your program remains in control, and can branch to different execution paths based on the outcome of every screen interaction.

On the other hand, however, if every aXes terminal operation was to be executed this way, the amount and complexity of the code in your application may become excessive in a non-trivial implementation. In addition, the number of JSM commands that your application needs to execute may begin to adversely affect the performance of your solution.

To enable you to choose the appropriate balance amongst the considerations of flexibility, complexity and performance, the AxesTerminalService provides support for aXes terminal operations scripts. This support is provided through two specialized service commands:

- The **LOADSCRIPT** command loads an aXes terminal operation script into the service for use in subsequent EXECUTE commands.
- The **EXECUTE** command executes a specified routine in an aXes terminal operation script.

Using this support, your application can replace a series of SETBYNAME or SETBYPOS commands that populate one entry screen, for example, with a single EXECUTE command. You can retain the degree of control necessary for your application and branch based on the outcome of 5250 screen operations by (optionally) breaking up your script into separate routines that are executed separately. For example, you may define a script routine to complete each non-trivial 5250 data entry screen. For a data entry screen containing a subfile, you might define a routine to complete each subfile record or line.

Typically, an implementation would define one script (optionally consisting of multiple routines), load the script once using the LOADSCRIPT command and execute the routines as required using the EXECUTE command. However, you may load and execute the scripts using the EXECUTE command if you prefer.

You may also mix this approach with direct use of the other service commands such as SETBYNAME, SETBYPOS, and SEND.

The script is provided as a list (for a LANSAR DML/X client, a working list) that is passed to the service with the LOADSCRIPT and/or EXECUTE commands. The list must contain a minimum of eight fields/columns and may contain up to twelve. The purpose and meaning of the list fields/columns that make up an aXes terminal operations script is described below.

1. Routine name Specifies the name of a routine to which this aXes terminal operations script line belongs. Lines for a routine must be specified contiguously in the list. The routine name is optional in the script and in the ROUTINE keyword of the EXECUTE command.
2. Label name This column may specify a unique (within a routine) label associated with the line. This label may be referenced as the next action on success or failure of other lines in the same routine/script.
3. Operation Specifies the aXes terminal operation performed by this script line. It may be one of the following:
 - GET (gets a field value)
 - SET (sets a field value)
 - SEND (sends data to the aXes terminal server)Lines containing any other value in the Operation column are ignored.
4. Field name (GET/SET) This column may specify the name of the field whose value is to be set or retrieved. The name may be a user-defined name (if an aXes project name was specified on the CONNECT command) or an aXes field identifier. If the field name is not specified for a GET or SET operation, then the field row and column must be specified.
5. Field index (GET/SET) If the field named in the previous column is contained in a subfile, this column may specify the index of the instance on the current screen of the field whose value is to be set or retrieved. The first instance on the current screen has an index of 1, irrespective of the scroll position of the subfile. If your program needs to process fields in subfiles, it must do so

one screen at a time and send ROLL_UP/ROLL_DOWN using the SEND command (just like a 5250 terminal user would have to do).

6. Field row (GET/SET) This column may specify the screen row number of the field whose value is to be set or retrieved. The row and column together, if used, must refer to the first position of the required field. The value of this column is ignored if the field name is specified. Conversely, if the field name is not specified, then the field row and column must be specified.
7. Field column (GET/SET) This column may specify the screen column number of the field whose value is to be set or retrieved. The row and column together, if used, must refer to the first position of the required field. The value of this column is ignored if the field name is specified. Conversely, if the field name is not specified, then the field row and column must be specified.
8. Field value (literal) (SET) This column may specify the literal value to be set in the specified field.
9. Field value (field name) (GET/SET) This column may specify the name of a field exchanged between your client program and the JSM that will receive (GET) or provide (SET) the value for the specified screen field. You will usually need to specify SERVICE_EXCHANGE(*FIELD) in the command string in order to receive the return value into your client program variable.
For a GET operation, the exchange field name is required.
For a SET operation, you must specify one of the exchange field names in this column or the literal field value in the previous column.
10. Function key (SET/SEND) If a value is specified in this column, the operation will send the current aXes screen data to the aXes terminal server using the function key specified (after successfully setting the field value for a SET operation). You may use any of the function key values shown for the SETBYNAME, SETBYPOS and SEND commands.
For a SET operation, if no value is specified, no send

operation is performed.

For a SEND operation, if no value is specified, ENTER is assumed.

11. Action on success This column may be used to control the next script routine operation performed when this operation completes successfully. It may be one of the following:
- *NEXT (the script proceeds to the next line in the same routine)
 - *ERROR (the script execution ends with an error status).
- Any other non-blank value is interpreted as the Label name of another script line in the same script routine, and the script continues with the first line in the script routine that matches the specified label.
- If no value is specified, *NEXT is assumed.
12. Action on error This column may be used to control the next script routine operation performed when this operation ends in error. It may be one of the following:
- *NEXT (the script proceeds to the next line in the same routine)
 - *ERROR (the script execution ends with an error status).
- Any other non-blank value is interpreted as the Label name of another script line in the same script routine, and the script continues with the first line in the script routine that matches the specified label.
- If no value is specified, *ERROR is assumed.

5.41.4 AxesTerminalService Commands

Your application issues commands to the AxesTerminalService by passing the command strings through the Java Services Manager using the JSM_COMMAND or JSMX_COMMAND built-in function or an API for your chosen development language.

The commands that the AxesTerminalService processes are:

SERVICE_LOAD
CONNECT
DISCONNECT
CAPTURE
EXECUTE
GET
GETBYNAME
GETBYPOS
GETFIELD
LOADSCRIPT
SEND
SETBYNAME
SETBYPOS
SETCURSOR
SERVICE_GET
SERVICE_UNLOAD

SERVICE_LOAD

Loads and initializes the service.

Required

```
SERVICE_LOAD ---- SERVICE ----- service name -----  
---->
```

Optional

```
>-- TRACE ----- *NO ----->  
      *YES  
      *ERROR  
  
>-- TRACE_NAME ----- name ----->  
      *SERVICE  
      *PROCESS  
      *FUNCTION  
      *JOBNAME  
      *JOBUSER  
      *JOBNUMBER  
  
>-- TRACESCREENS ---- *YES ----->  
      *NO  
  
>-- AXESLOG ----- *NO -----|  
      *YES
```

For more information refer to:

[SERVICE_LOAD](#)

[Service Program Tracing from the Client](#)

Keywords

SERVICE	The name of the service to be loaded - in this case AxesTerminalService.
---------	--

- TRACE** To enable tracing from the client program use the TRACE keyword on the SERVICE_LOAD command. The possible values for the TRACE keyword are:
- *NO
 - *YES
 - *ERROR
- The trace option *ERROR will turn on tracing and if the service does not return an ERROR, FATAL or SOAPFAULT status to the client, the trace files are deleted on the JSM CLOSE call.
- The TRACE keyword will override the settings in the manager.properties file.
-
- TRACE_NAME** This optional keyword allows the client to append a user-defined name to the end of the client trace subdirectory. Special keyword values are also available for the TRACE_NAME keyword.
- *SERVICE
 - *PROCESS
 - *FUNCTION
 - *JOBNAME
 - *JOBUSER
 - *JOBNUMBER
-
- TRACESCREENS** If tracing is in effect (according to the value specified on the TRACE keyword or in the manager.properties file), this keyword may be used to enable additional tracing of the aXes terminal screen images. Note that enabling the tracing of the screen images may greatly expand the size of the resulting service.txt trace file.
- The possible values for the TRACESCREENS keyword are:
- *YES
 - *NO
- If not specified, a default of *YES is assumed. Note that

this is only effective if tracing is enabled.

AXESLOG

If tracing is in effect (according to the value specified on the TRACE keyword or in the manager.properties file), this keyword may be used to enable the creation of an aXes log file. The aXes log file may be useful in diagnosing issues associated with the aXes terminal operations.

The possible values for the AXESLOG keyword are:

- *NO
- *YES

If not specified, a default of *NO is assumed.

Examples

RDML Example:

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(AXESTERM  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
#jsmcommand := 'service_load service(AxesTerminalService)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcommand) to_get(#jsn
```

CONNECT

The CONNECT command is used to connect to an aXes server and signon the 5250 session using the server identification and user credentials provided.

Note: You usually need to specify SERVICE_EXCHANGE(*FIELD) in the command string in order to receive the return values into your client program variables named on keywords beginning with RTNxxx. Refer to [5.41.2 Using the AxesTerminalService](#) for more information.

Required

CONNECT ----- *HOST* ----- *value* ----->

>-- *USER* ----- *value* ----->

>-- *PASSWORD* ----- *value* ----->

Optional

>-- *PORT* ----- 80 ----->
value

>-- *USESSL* ----- *NO ----->
**YES*

>-- *PROJECT* ----- *value* ----->

>-- *RTNSIG* ----- *field-name* ----->

>-- *RTNSCREEN* ----- *field-name* -----|

Keywords

HOST Required. The host name or IP address of the machine hosting the aXes terminal server to which you wish to connect.

USER	Required. The user name used to connect to the aXes terminal server. This must be the name of an authorised i/OS user profile.
PASSWORD	Required. The password for the user name specified in the USER keyword.
PORT	The port number to be used to connect to the aXes terminal server. If not specified, a default value of 80 is assumed.
PROJECT	The aXes project to use for this session. A project contains user defined definitions of screens and fields. For example, you can assign a name to a screen or a field, and later use that name to refer to the screen or the field in your code. Although aXes projects can either be stored in a file on the file system or on the aXes server, you may only specify here an aXes server-based project.
RTNSIG	The name of a field exchanged between your client program and the JSM that will receive the screen signature of the screen that is present following the successful connection.
RTNSCREEN	The name of a field exchanged between your client program and the JSM that will receive the name of the screen that is present following the successful connection. The screen name is the user-defined name if an aXes project name was specified on the CONNECT command and a name was assigned to the screen. Otherwise the screen signature is returned.

Examples

RDML Example:

```
DEFINE FIELD(#MYSCREEN) REFFLD(#STD_TEXT)
CHANGE FIELD(#JSMCMD) TO('CONNECT HOST(MYSERVER) USER(
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
define field(#myscreen) reffld(#std_text)
#jasmcommand := 'CONNECT HOST(MYSERVER) USER(QUSER) PASSW(
use builtin(jsmx_command) with_args(#jasmhandle #jasmcommand) to_get(#jsn
```


DISCONNECT

The DISCONNECT command disconnects the service from the aXes Terminal Server.

Required

DISCONNECT -----|

Keywords

There are no keywords for the DISCONNECT command.

Examples

RDML Example:

```
CHANGE FIELD(#JSMCMD) TO('DISCONNECT'  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
#jasmcommand := 'DISCONNECT'  
use builtin(jsmx_command) with_args(#jasmhandle #jasmcommand) to_get(#jsn
```

CAPTURE

The CAPTURE command captures a representation of the current aXes Terminal Server screen and returns it in the list provided.

Optional

```
CAPTURE ----- RTNSIG ----- field-name -----  
>
```

```
>-- RTNSCREEN ----- field-name -----|
```

You must provide a list definition containing at least one alphanumeric field in which the captured screen lines are returned. The service command will fill the list with an entry for each screen line captured. The captured screen text will be placed in the first field in the list, which is the only field used.

Keywords

- RTNSIG** The name of a field exchanged between your client program and the JSM that will receive the screen signature of the screen that is present.
- RTNSCREEN** The name of a field exchanged between your client program and the JSM that will receive the name of the screen that is present. The screen name is the user-defined name if an aXes project name was specified on the CONNECT command and a name was assigned to the screen. Otherwise the screen signature is returned.

Examples

RDML Example:

```
DEFINE FIELD(#MYSCREEN) REFFLD(#STD_TEXT)  
DEFINE FIELD(#SCRNLIN) TYPE(*CHAR) LENGTH(80)  
DEF_LIST NAME(#CAPTURE) FIELDS(#SCRNLIN) TYPE(*WORKING  
CHANGE FIELD(#JSMCMD) TO('CAPTURE RTNSCREEN(MYSCREEN)  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
define field(#myscreen) reffld(#std_text)
define field(#scrnline) type(*char) length(80)
def_list name(#capture) fields(#scrnline) type(*working)
#jasmcommand := 'CAPTURE RTNSCREEN(MYSCREEN) SERVICE_EXCH
use builtin(jsmx_command) with_args(#jasmhandle #jasmcommand) to_get(#jsn
```

EXECUTE

The EXECUTE command executes a specified routine in an aXes terminal operation script. Refer to [5.41.3 AxesTerminalService Scripts](#) for more information about specifying and using aXes terminal operation scripts.

Optional

```
EXECUTE ----- SCRIPT ----- *LOADED -----  
--->  
      *LIST  
  
>-- ROUTINE ----- routine-name ----->  
  
>-- RTNSIG ----- field-name ----->  
  
>-- RTNSCREEN ----- field-name -----|
```

If the command specifies SCRIPT(*LIST), you must provide a list definition that contains the aXes terminal operation script. Refer to [5.41.3 AxesTerminalService Scripts](#) for more information about specifying and using aXes terminal operation scripts.

Keywords

- SCRIPT** How the aXes terminal operation script is provided. It may contain one of the following values:
- *LOADED : the script has previously been provided by means of the LOADSCRIPT command.
 - *LIST : the script is provided in a working list with this command.
- If not specified, the service assumes SCRIPT(*LOADED).
- ROUTINE** The name of a routine in the aXes terminal operation script. Refer to [5.41.3 AxesTerminalService Scripts](#) for more information about specifying and using routines in aXes terminal operation scripts.
- If not specified, a blank routine name is assumed. If your

script does not contain named routines you do not need to specify this keyword.

RTNSIG The name of a field exchanged between your client program and the JSM that will receive the screen signature of the screen that is present following execution of the aXes terminal operation script.

RTNSCREEN The name of a field exchanged between your client program and the JSM that will receive the name of the screen that is present following execution of the aXes terminal operation script. The screen name is the user-defined name if an aXes project name was specified on the CONNECT command and a name was assigned to the screen. Otherwise the screen signature is returned.

Examples

RDML Example:

```
DEFINE FIELD(#MYSCREEN) REFFLD(#STD_TEXT)
CHANGE FIELD(#JSMCMD) TO('EXECUTE ROUTINE(ENTERORDER) I
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
define field(#myscreen) reffld(#std_text)
#jasmcommand := 'EXECUTE ROUTINE(ENTERORDER) RTNSCREEN(MY
use builtin(jsmx_command) with_args(#jasmhandle #jasmcommand) to_get(#jsn
```

GET

The client application may issue the GET command to receive attributes associated with the aXes terminal session into named client program variables.

Note: You usually need to specify SERVICE_EXCHANGE(*FIELD) in the command string in order to receive the return values into your client program variables named on keywords beginning with RTNxxx. Refer to [5.41.2 Using the AxesTerminalService](#) for more information.

Optional

```
GET ----- RTNCURSORROW ---- field-name -----  
-->  
  
  >-- RTNCURSORCOL ---- field-name ----->  
  
  >-- RTNDEVICENAME --- field-name ----->  
  
  >-- RTNJOBUSER ----- field-name ----->  
  
  >-- RTNJOBNUMBER ---- field-name ----->  
  
  >-- RTNSCREENWIDTH -- field-name ----->  
  
  >-- RTNSCREENHEIGHT - field-name -----  
>  
  
  >-- RTNSYSTEMNAME --- field-name ----->  
  
  >-- RTNSIG ----- field-name ----->  
  
  >-- RTNSCREEN ----- field-name -----|
```

Keywords

RTNCURSORROW The name of a field exchanged between your client program and the JSM that will receive the row

	number component of the current screen cursor position.
RTNCURSORCOL	The name of a field exchanged between your client program and the JSM that will receive the column number component of the current screen cursor position.
RTNDEVICENAME	The name of a field exchanged between your client program and the JSM that will receive the 5250 device name used by the current aXes terminal session.
RTNJOBUSER	The name of a field exchanged between your client program and the JSM that will receive the user name portion of the i/OS job name for the current aXes terminal session.
RTNJOBNUMBER	The name of a field exchanged between your client program and the JSM that will receive the job number portion of the i/OS job name for the current aXes terminal session.
RTNSCREENWIDTH	The name of a field exchanged between your client program and the JSM that will receive the current screen width for the aXes terminal session. Note that this value refers to the screen currently displayed, not maximum device capabilities.
RTNSCREENHEIGHT	The name of a field exchanged between your client program and the JSM that will receive the current screen height for the aXes terminal session. Note that this value refers to the screen currently displayed, not maximum device capabilities.
RTNSYSTEMNAME	The name of a field exchanged between your client program and the JSM that will receive the i/OS system name for the current aXes terminal session.
RTNSIG	The name of a field exchanged between your client program and the JSM that will receive the screen signature of the screen that is present.

RTNSCREEN

The name of a field exchanged between your client program and the JSM that will receive the name of the screen that is present. The screen name is the user-defined name if an aXes project name was specified on the CONNECT command and a name was assigned to the screen. Otherwise the screen signature is returned.

Examples

RDML Example:

```
DEFINE FIELD(#MYJOBUSR) REFFLD(#STD_TEXT)
DEFINE FIELD(#MYJOBNUM) REFFLD(#STD_TEXT)
CHANGE FIELD(#JSMCMD) TO('GET RTNJOBUSER(MYJOBUSR) RTNJ
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
define field(#myjobusr) reffld(#std_text)
define field(#myjobnum) reffld(#std_text)
#jasmcommand := 'GET RTNJOBUSER(MYJOBUSR) RTNJOBNUMBER(M
use builtin(jsmx_command) with_args(#jasmhandle #jasmcommand) to_get(#jsn
```

GETBYNAME

The GETBYNAME command retrieves the value of a field on the current aXes terminal screen by reference to the field name (and an optional index for a subfile field).

Note: You usually need to specify SERVICE_EXCHANGE(*FIELD) in the command string in order to receive the return value into your client program variable named on the RTNVALUE keyword. Refer to [5.41.2 Using the AxesTerminalService](#) for more information.

Required

```
GETBYNAME ----- NAME ----- screen field name -----  
-->
```

```
>-- RTNVALUE ----- field-name ----->
```

Optional

```
>-- INDEX ----- 0 -----|  
value
```

Keywords

- | | |
|-------|--|
| NAME | Required. The name of the field whose value is to be retrieved. The name may be a user-defined name (if an aXes project name was specified on the CONNECT command) or an aXes field identifier. |
| INDEX | If the field is contained in a subfile, this keyword may specify the index of the instance on the current screen of the field whose value is to be retrieved. The first instance on the current screen has an index of 1, irrespective of the scroll position of the subfile. If your program needs to process fields in subfiles, it must do so a screen at a time and send ROLL_UP/ROLL_DOWN using the SEND command (just like a 5250 terminal user would have to do). |

RTNVALUE The name of a field exchanged between your client program and the JSM that will receive the value for the specified screen field.

Examples

RDML Example:

```
DEFINE FIELD(#MYCUSNUM) REFFLD(#STD_TEXT)
CHANGE FIELD(#JSMCMD) TO('GETBYNAME NAME(FLD001) RTNVA
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
define field(#mycusnum) reffld(#std_text)
#jasmcommand := 'GETBYNAME NAME(FLD001) RTNVALUE(MYCUSNU
use builtin(jsmx_command) with_args(#jasmhandle #jasmcommand) to_get(#jsn
```

GETBYPOS

The GETBYPOS command retrieves the value of a field on the current aXes terminal screen by reference to the screen row and column number.

Note: You usually need to specify SERVICE_EXCHANGE(*FIELD) in the command string in order to receive the return value into your client program variable named on the RTNVALUE keyword. Refer to [5.41.2 Using the AxesTerminalService](#) for more information.

Required

```
GETBYPOS ----- ROW ----- row number -----  
>  
  
  >-- COL ----- column number ----->  
  
  >-- RTNVALUE ----- field-name -----|
```

Keywords

- | | |
|----------|---|
| ROW | Required. The screen row number of the required field. The row and column together must refer to the first position of the required field. |
| COL | Required. The screen column number of the required field. The row and column together must refer to the first position of the required field. |
| RTNVALUE | The name of a field exchanged between your client program and the JSM that will receive the value for the specified screen field. |

Examples

RDML Example:

```
DEFINE FIELD(#MYCUSNUM) REFFLD(#STD_TEXT)  
CHANGE FIELD(#JSMCMD) TO('GETBYPOS ROW(8) COL(44) RTNVAL  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
define field(#mycusnum) reffld(#std_text)
#jsmcommand := 'GETBYPOS ROW(8) COL(44) RTNVALUE(MYCUSNUM
use builtin(jsmx_command) with_args(#jsmhandle #jsmcommand) to_get(#jsn
```

GETFIELD

The GETFIELD command retrieves the attributes of a field on the current aXes terminal screen by reference to either the field name (and an optional index for a subfile field) or to the screen row and column number of the field.

Note: You usually need to specify SERVICE_EXCHANGE(*FIELD) in the command string in order to receive the return values into your client program variables named on keywords beginning with RTNxxx. Refer to [5.41.2 Using the AxesTerminalService](#) for more information.

Optional

GETFIELD ----- *NAME* ----- *screen field name* -----

>

>-- *INDEX* ----- *0* ----->
value

>-- *ROW* ----- *row number* ----->

>-- *COL* ----- *column number* ----->

RTNNAME ----- *field-name* ----->

>-- *RTNINDEX* ----- *field-name* ----->

>-- *RTNROW* ----- *field-name* ----->

>-- *RTNCOL* ----- *field-name* ----->

>-- *RTNSUBFILEEXTENT* *field-name* -----|

Keywords

NAME

The name of the field whose attributes are to be retrieved. The name may be a user-defined name (if an aXes project name was specified on the CONNECT command) or an

aXes field identifier. If NAME is not specified, then ROW and COL must be specified.

INDEX	If the field is contained in a subfile, this keyword may specify the index of the instance on the current screen of the field whose attributes are to be retrieved. The first instance on the current screen has an index of 1, irrespective of the scroll position of the subfile. If your program needs to process fields in subfiles, it must do so a screen at a time and send ROLL_UP/ROLL_DOWN using the SEND command (just like a 5250 terminal user would have to do).
ROW	The screen row number of the required field. The row and column together, if used, must refer to the first position of the required field. The value of this keyword is ignored if NAME is specified. Conversely, if NAME is not specified, then ROW and COL must be specified.
COL	The screen column number of the required field. The row and column together, if used, must refer to the first position of the required field. The value of this keyword is ignored if NAME is specified. Conversely, if NAME is not specified, then ROW and COL must be specified.
RTNNAME	The name of a field exchanged between your client program and the JSM that will receive the user-defined name of the field (if an aXes project name was specified on the CONNECT command) or the aXes field identifier if no project was specified or no name has been defined for the field.
RTNINDEX	If the field is contained in a subfile, this keyword may be used to specify the name of a field exchanged between your client program and the JSM that will receive the index of the instance on the current screen of the field. The first instance on the current screen has an index of 1, irrespective of the scroll position of the subfile.
RTNROW	The name of a field exchanged between your client program and the JSM that will receive the screen row number of the field.

RTNCOL	The name of a field exchanged between your client program and the JSM that will receive the screen column number of the field.
RTNSU BFILEEXTENT	If the field is contained in a subfile, this keyword may be used to specify the name of a field exchanged between your client program and the JSM that will receive the number of instances on the current screen for the field (the number of subfile records displayed on the current screen). For reliability, you should use this keyword for a field that is never conditioned on the subfile display - such as the option entry field in a "Work with" style display.

Examples

RDML Example:

```
DEFINE FIELD(#MYSFLEXT) REFFLD(#STD_NUM)
CHANGE FIELD(#JSMCMD) TO('GETFIELD ROW(8) COL(44) RTNSUBFI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
define field(#mysflect) reffld(#std_num)
#jsmcommand := 'GETFIELD ROW(8) COL(44) RTNSUBFILEEXTENT(MY
use builtin(jsmx_command) with_args(#jsmhandle #jsmcommand) to_get(#jsn
```

LOADSCRIPT

The LOADSCRIPT command loads an aXes terminal operation script into the service for use in subsequent EXECUTE commands. The script replaces any previously loaded. Refer to [5.41.3 AxesTerminalService Scripts](#) for more information about specifying and using aXes terminal operation scripts.

Optional

*LOADSCRIPT ----- SCRIPT ----- *LIST -----*

If the command specifies SCRIPT(*LIST), you must provide a list definition that contains the aXes terminal operation script. Refer to [5.41.3 AxesTerminalService Scripts](#) for more information about specifying and using aXes terminal operation scripts.

Keywords

SCRIPT This keyword specifies how the aXes terminal operation script is provided. Presently only the following value is supported:
***LIST** : the script is provided in a working list with this command.
If not specified, the service assumes SCRIPT(*LIST).

Examples

RDML Example:

```
CHANGE FIELD(#JSMCMD) TO('LOADSCRIPT SCRIPT(*LIST)'  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
#jsmcommand := 'LOADSCRIPT SCRIPT(*LIST)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcommand) to_get(#jsn
```

SEND

The SEND command sends the current aXes screen data to the aXes terminal server. It is comparable to pressing the ENTER key (by default) or a function key at a 5250 terminal.

Note: You usually need to specify SERVICE_EXCHANGE(*FIELD) in the command string in order to receive the return values into your client program variables named on keywords beginning with RTNxxx. Refer to [5.41.2 Using the AxesTerminalService](#) for more information.

Optional

```
SEND ----- SENDKEY ----- ENTER -----  
>  
    F1 - F9  
    F10 - F24  
    HELP  
    PRINT  
    PAGE_UP  
    PAGE_DOWN  
    ROLL_UP  
    ROLL_DOWN  
  
>-- CURSORROW ----- cursor row number ----->  
  
>-- CURSORCOL ----- cursor column number -----  
>  
  
>-- RTNSIG -- ----- field-name ----->  
  
>-- RTNSCREEN ----- field-name -----|
```

Keywords

SENDKEY The function key to send. You may use any of the values shown above. If not specified, ENTER is sent.

- CURSROW** The cursor row number to send. If CURSROW and CURSORCOL are not both specified, the cursor position is unchanged.
- CURSORCOL** The cursor column number to send. If CURSROW and CURSORCOL are not both specified, the cursor position is unchanged.
- RTNSIG** The name of a field exchanged between your client program and the JSM that will receive the screen signature of the screen that is present following the send operation.
- RTNSCREEN** The name of a field exchanged between your client program and the JSM that will receive the name of the screen that is present following the send operation. The screen name is the user-defined name if an aXes project name was specified on the CONNECT command and a name was assigned to the screen. Otherwise the screen signature is returned.

Examples

RDML Example:

```
DEFINE FIELD(#MYSCREEN) REFFLD(#STD_TEXT)
CHANGE FIELD(#JSMCMD) TO('SEND SENDKEY(F3) RTNSCREEN(MY
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
define field(#myscreen) reffld(#std_text)
#jsmcommand := 'SEND SENDKEY(F3) RTNSCREEN(MYSCREEN) SERV
use builtin(jsmx_command) with_args(#jshandle #jsmcommand) to_get(#jsn
```

SETBYNAME

The SETBYNAME command sets the value of a field on the current aXes terminal screen by reference to the field name (and an optional index for a subfile field) and may (optionally) send the current aXes screen data to the aXes terminal server. It is comparable to typing at a 5250 terminal.

Note: You usually need to specify SERVICE_EXCHANGE(*FIELD) in the command string in order to receive the return values into your client program variables named on keywords beginning with RTNxxx. Refer to [5.41.2 Using the AxesTerminalService](#) for more information.

Required

```
SETBYNAME ----- NAME ----- screen field name -----  
->
```

```
>-- VALUE ----- value ----->
```

Optional

```
>-- INDEX ----- 0 ----->  
value
```

```
>-- SENDKEY -----(none) ----->
```

```
ENTER  
F1 ... F9  
F10 ... F24  
HELP  
PRINT  
PAGE_UP  
PAGE_DOWN  
ROLL_UP  
ROLL_DOWN
```

```
>-- RTNSIG -- ----- field-name ----->
```

```
>-- RTNSCREEN ----- field-name -----|
```

Keywords

NAME	This keyword must specify the name of the field whose value is to be set. The name may be a user-defined name (if an aXes project name was specified on the CONNECT command) or an aXes field identifier.
INDEX	If the field is contained in a subfile, this keyword may specify the index of the instance on the current screen of the field whose attributes are to be retrieved. The first instance on the current screen has an index of 1, irrespective of the scroll position of the subfile. If your program needs to process fields in subfiles, it must do so a screen at a time and send ROLL_UP/ROLL_DOWN using the SEND command (just like a 5250 terminal user would have to do).
VALUE	This keyword must specify the value to be set in the specified field.
SENDKEY	If this keyword is specified, the command will send the current aXes screen data to the aXes terminal server using the function key specified (after successfully setting the field value). You may use any of the function key values shown above. If not specified, no send operation is performed.
RTNSIG	The name of a field exchanged between your client program and the JSM that will receive the screen signature of the screen that is present following the operation.
RTNSCREEN	The name of a field exchanged between your client program and the JSM that will receive the name of the screen that is present following the operation. The screen name is the user-defined name if an aXes project name was specified on the CONNECT command and a name was assigned to the screen. Otherwise the screen signature is returned.

Examples

RDML Example:

```
DEFINE FIELD(#MYSCREEN) REFFLD(#STD_TEXT)
CHANGE FIELD(#JSMCMD) TO('SETBYNAME NAME(FLD001) VALUE
```

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS

RDMLX Example:

```
define field(#myscreen) reffld(#std_text)
#jasmcommand := 'SETBYNAME NAME(FLD001) VALUE(ABC001) RTNSC
use builtin(jsmx_command) with_args(#jasmhandle #jasmcommand) to_get(#jsn
```

SETBYPOS

The SETBYPOS command sets the value of a field on the current aXes terminal screen by reference to the screen row and column number and may (optionally) send the current aXes screen data to the aXes terminal server. It is analogous to typing at a 5250 terminal.

Note: You usually need to specify SERVICE_EXCHANGE(*FIELD) in the command string in order to receive the return values into your client program variables named on keywords beginning with RTNxxx. Refer to [5.41.2 Using the AxesTerminalService](#) for more information.

Required

SETBYPOS ----- *ROW* ----- *row number* -----
>

>-- *COL* ----- *column number* ----->

>-- *VALUE* ----- *value* ----->

Optional

>-- *SENDKEY* -----(*none*) ----->

ENTER
F1 ... F9
F10 ... F24
HELP
PRINT
PAGE_UP
PAGE_DOWN
ROLL_UP
ROLL_DOWN

>-- *RTNSIG* -- ----- *field-name* ----->

>-- *RTNSCREEN* ----- *field-name* -----|

Keywords

ROW	Required. The screen row number of the required field. The row and column together must refer to the first position of the required field.
COL	Required. The screen column number of the required field. The row and column together must refer to the first position of the required field.
VALUE	Required. The value to be set in the specified field.
SENDKEY	If this keyword is specified, the command will send the current aXes screen data to the aXes terminal server using the function key specified (after successfully setting the field value). You may use any of the function key values shown above. If not specified, no send operation is performed.
RTNSIG	The name of a field exchanged between your client program and the JSM that will receive the screen signature of the screen that is present following the operation.
RTNSCREEN	The name of a field exchanged between your client program and the JSM that will receive the name of the screen that is present following the operation. The screen name is the user-defined name if an aXes project name was specified on the CONNECT command and a name was assigned to the screen. Otherwise the screen signature is returned.

Examples

RDML Example:

```
DEFINE FIELD(#MYSCREEN) REFFLD(#STD_TEXT)
CHANGE FIELD(#JSMCMD) TO('SETBYPOS ROW(8) COL(44) VALUE(ABC001) RTNSIG
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMHANDLE #JSMCMD)
```

RDMLX Example:

```
define field(#myscreen) reffld(#std_text)
#jsmcommand := 'SETBYPOS ROW(8) COL(44) VALUE(ABC001) RTNSIG
use builtin(jsmx_command) with_args(#jsmhandle #jsmcommand) to_get(#jsmhandle #jsmcommand)
```

SETCURSOR

The SETCURSOR command sets the position of the cursor on the current aXes terminal screen. It is analogous to using the arrow or tab keys to move the cursor at a 5250 terminal. Some 5250 applications or screens are sensitive to the cursor position and may exhibit different functionality dependent upon it.

Required

```
SETCURSOR ----- CURSORROW ----- cursor row number --  
----->  
  
>-- CURSORCOL ----- cursor column number -----|
```

Keywords

CURSORROW Required. The cursor row number to set.

CURSORCOL Required. The cursor column number to set.

Examples

RDML Example:

```
CHANGE FIELD(#JSMCMD) TO('SETCURSOR CURSORROW(8) CURSC  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

RDMLX Example:

```
#jsmcommand := 'SETCURSOR CURSORROW(8) CURSORCOL(44)  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcommand) to_get(#jsn
```

SERVICE_GET

The SERVICE_GET command allows JSM client applications to access the keyword / value properties from the properties file associated with the currently loaded service.

Client applications can access the value of properties associated with the service, but, perhaps more usefully, they can access application specific property values too, using the service properties file as a common medium for holding such values.

When used in the latter way, there is a possibility of conflict between application-defined keyword names and those used by current and future versions of LANSA Integrator. To reduce this possibility your application should prefix the property keyword with a global unique name. The examples below use com.acme.property as the prefix for this purpose.

The service property keyword value is returned in the JSM message field. If the keyword does not exist, then a status of NOT_EXIST is returned.

Required

```
SERVICE_GET ----- PROPERTY ----- value -----  
-|
```

Keywords

PROPERTY Provide a keyword / value to be checked on the service properties file.

The service property keyword value is returned in the JSM message field. If the keyword does not exist in the service property file a status of NOT_EXIST is returned.

Examples

The examples retrieve the value of the com.acme.property.message type (an application-defined property) in the service properties file. This property might be defined in the service properties file with a line like this (the value "html" will be returned to the application in this case):

```
com.acme.property.message=html
```

RDML Example:

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_GET PROPER'
```

RDMLX Example:

```
#jsmcmd := ' service_get property(com.acme.property.messageType)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service and removes the temporary directory.

```
SERVICE_UNLOAD ----- no keywords -----  
-----|
```

Keywords

There are no SERVICE_UNLOAD keywords.

Examples

RDML Example:

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO
```

RDMLX Example:

```
use builtin(jsmx_command) with_args(#jsmhandle service_unload) to_get(#jsr
```

5.41.5 AxesTerminalService Examples

This example is supplied for the most common client programming language:

[RDMLX](#)

5.42 HashService

Service Name: **HashService**

The HashService can generate a hash from a specified field value or specified file contents.

The HashService supports the following commands:

5.42.1 [SERVICE_LOAD](#)

5.42.2 [SERVICE_GET](#)

5.42.3 [HASH](#)

5.42.4 [SERVICE_UNLOAD](#)

5.42.1 SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD			
--------------	--	--	--

5.42.2 SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

5.42.3 HASH

The HASH command generates the hash of the specified field or file. The hash value is returned in the command message response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
HASH	FILE	value	Optional. File Path.
	FIELD	value	Optional. Field Name.
	DIGEST	value	Optional. Hashing algorithm supported by the JVM. MD2,MD5,SHA,SHA256,SHA384,SHA512 Default. SHA.
	ENCODING	value	Optional. Field value to byte encoding. Default. UTF-8.
	HASHENCODING	*HEX *BASE64	Optional. String encoding of binary data. Default.

5.42.4 SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_UNLOAD			

6. Create Your Own Services

If you have already created a set of Java classes or if you wish to use some third party Java classes, you will need to write your own service for the Java Service Manager.

Once the service has been created and deployed, it can be called by your LANSAs function or RPG ILE program.

Before you create your own service, it is recommended that you review:

- [6.1 Create your Custom Service](#)
- [6.2 Java Service Manager Interfaces](#)
- [6.3 Java Service Manager Classes](#)

6.1 Create your Custom Service

Your custom service needs to implement the JSMService interface.

Your development environment needs to have access to jsml.jar, jsmservice.jar and jsmutil.jar.

Create and compile your custom service.

Put your custom service class in the package directory below the classes directory.

```
/jsm/instance/classes/com/lansa/jsm/userdefined/XYZService.class
```

Add an entry to service.properties in the system directory.

```
service.XYZService=com.lansa.jsm.userdefined.XYZService
```

Note: Standard LANSA Integrator licences that permit the use of user-defined services, require that the custom service class package name uses the **com.lansa.jsm.userdefined.** prefix. Unless you make special licencing arrangements, your custom services must use this naming convention.

Create a properties file called XYZService.properties in the properties directory.

```
#!<studio-project id="20050606-115520" name="XYZ_Project">  
# XYZService resource ( Default )  
message.001=XYZService successfully loaded  
message.002=XYZService successfully unloaded  
message.003=Command is not supported :  
#!</studio-project>
```

Sample LANSA function

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_LOAD(XYZSE  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(SERVICE_UNLOAD) TO  
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

Sample XYZService code

```
package com.lansa.jsm.userdefined ;

import java.io.* ;
import java.net.* ;
import java.util.* ;

import com.lansa.jsm.* ;

public final class XYZService implements JSMService
{
    private JSMTrace m_trace = null ;
    private JSMResource m_serviceResource = null ;

    public XYZService ()
    {
        if ( !JSMManager.isLicenced ( this ) )
        {
            throw new IllegalArgumentException ( "Class is not licenced : " + this.
        }
    }

    public final void service ( JSMContainer container )
    {
        m_trace = container.getServiceTrace () ;

        m_serviceResource = container.getServiceResource () ;
    }

    public final JSMResponse command ( JSMCommand command ) throws JS
    {
        try
        {
            if ( m_trace != null )
            {
                m_trace.print ( command ) ;
            }
        }
    }
}
```

```

    }

    JSMResponse response = runCommand ( command ) ;

    if ( m_trace != null )
    {
        m_trace.print ( command, response ) ;
    }

    return response ;
}
catch ( Throwable t )
{
    if ( m_trace != null )
    {
        m_trace.print ( command, t ) ;
    }

    return new JSMResponse ( t ) ;
}
}

```

```

private final JSMResponse runCommand ( JSMCommand command ) throw
Exception
{
    if ( command.equals ( JSMCommand.SERVICE_LOAD ) )
    {
        return commandLOAD ( command ) ;
    }

    if ( command.equals ( JSMCommand.SERVICE_UNLOAD ) )
    {
        return commandUNLOAD ( command ) ;
    }

    if ( command.equals ( "SEND" ) )
    {
        return commandSEND ( command ) ;
    }
}

```

```
        return new JSMResponse ( JSMResponse.ERROR, m_serviceResource.g
    }

private final JSMResponse commandLOAD ( JSMCommand command ) th
{
    return new JSMResponse ( m_serviceResource.getResource ( "message.0
}

private final JSMResponse commandUNLOAD ( JSMCommand command )
{
    return new JSMResponse ( m_serviceResource.getResource ( "message.0
}

private final JSMResponse commandSEND ( JSMCommand command ) thr
{
    return new JSMResponse ( "" );
}
}
```

6.2 Java Service Manager Interfaces

The Java Service Manager is shipped with the following interfaces:

[6.2.1 JSMService Interface](#)

[6.2.2 JSMContainer Interface](#)

[6.2.3 JSMCommand Interface](#)

[6.2.4 JSMClient Interface](#)

[6.2.5 JSMResource Interface](#)

[6.2.6 JSMStorage Interface](#)

[6.2.7 JSMTrace Interface](#)

[6.2.8 JSMList Interface](#)

[6.2.9 JSMFieldList Interface](#)

[6.2.10 JSMField Interface](#)

6.2.1 JSMSERVICE Interface

For a Java class to be loaded and successfully executed, it must implement the JSMSERVICE interface as shown in this example:

```
public interface JSMSERVICE
{
    public void service ( JSMContainer container ) ;

    public JSMSResponse command ( JSMCommand command ) throws JSMExcpt
}
```

The service method of the class is called when the service first loads. This method will only be called once, so the JSMContainer object should be saved for later access in the current service program. The JSMContainer object allows access to the service trace object, service resource object and service storage object. The container resource is currently not used.

The command method is called every time the JSM client program sends a command string.

The command object allows access to the command string in Unicode, helper methods allow access to keywords. The client program byte encoding and [locale](#) can be accessed from the command object. Also a byte array object is available from the command object. This byte array is never null and will be a zero length array or a copy of the byte array sent from the client. No byte code transformation is done, so the byte array is raw and reflects the same encoding as the client.

6.2.2 JSMContainer Interface

```
public interface JSMContainer
{
    public JSMTrace getServiceTrace () ;
    public JSMStorage getServiceStorage () ;
    public JSMResource getServiceResource () ;
    public JSMResource getContainerResource () ;
}
```

The JSMContainer object allows the service program to gain information about its environment.

The JSMStorage object allows the service program to store objects. This storage will be valid across a SERVICE_LOAD and SERVICE_UNLOAD command. The storage object is valid between a JSM open and close. When the JSM closes the storage object is lost. The storage object is never null.

The JSMTrace object allows the service program to output tracing information. If the trace object is null, then tracing has been disabled.

The JSMResource object returned from the getServiceResource method is the resource file associated with the service program. This resource object is never null, if a resource properties file has not been assigned or does not exist then an empty resource object is supplied.

The JSMResource object returned from the getContainerResource is not used yet.

6.2.3 JSMCommand Interface

```
public interface JSMCommand
{
    public final static String SERVICE_LOAD    = "SERVICE_LOAD" ;
    public final static String SERVICE_UNLOAD = "SERVICE_UNLOAD" ;
    public final static String SERVICE_READ   = "SERVICE_READ" ;
    public final static String SERVICE_GET    = "SERVICE_GET" ;
    public final static String SERVICE_SET    = "SERVICE_SET" ;
    public final static String SERVICE_RECLAIM = "SERVICE_RECLAIM" ;

    public final static int HTTP_CONTEXT_UNKNOWN = 0 ;
    public final static int HTTP_CONTEXT_NONE   = 1 ; // Interactive
    public final static int HTTP_CONTEXT_CONNECT = 2 ; // HTTP connect
    public final static int HTTP_CONTEXT_REQUEST = 3 ; // HTTP request
    public final static int HTTP_CONTEXT_KEYWORDS = 4 ; // HTTP
keywords

    public String getCommand () ;

    public JSMList getList () ;
    public JSMFieldList getFieldList () ;

    public JSMClient getClient () ;

    public byte[] getByteArray () ;

    public Enumeration getKeywords () ;
    public String getKeywordValue ( String key ) ;

    public boolean isHTTP () ;
    public int getHTTPContext () ;

    public Enumeration getHTTPKeywords () ;
    public String getHTTPKeywordValue ( String key ) ;

    public boolean equals ( String command ) ;
}
```

}

- The command object is the Java object equivalent of the command string sent by the JSM client.
- The String object returned from the getCommand method is the command name.
- The equals method can be used to check the name of a command.
- The byte array object returned by the getByteArray method is the byte array sent by the JSM client.
- The JSMList object returned by the getList method is the working list sent by the JSM client.
- The JSMFieldList object returned by the getFieldList is the field list sent by the JSM client.
- The Enumeration object returned by the getKeywords method is an enumeration of the command keywords.
- The String value returned from the getKeywordValue method is the value associated with the keyword.
- The Enumeration object returned by the getHTTPKeywords method is an enumeration of the HTTP keywords.
- The String value returned from the getHTTPKeywordValue method is the value associated with the HTTP keyword.

For more information, refer to [Command](#).

6.2.4 JSMClient Interface

```
public interface JSMClient
{
    public String getType () ;
    public Locale getLocale () ;
    public String getVersion () ;
    public String getAddress () ;
    public String getEncoding () ;

    public String getDataLibrary () ;
    public String getModuleLibrary () ;
    public String getProgramLibrary () ;

    public String getProcess () ;
    public String getFunction () ;
    public String getPartition () ;
    public String getLanguage () ;

    public boolean isLanguageDBCS () ;
    public boolean isLanguageLeftToRight () ;
    public boolean isLanguageRightToLeft () ;

    public String getDateFormat () ;
    public String getDateSeparator () ;
    public String getTimeSeparator () ;

    public String getJobType () ;
    public String getJobName () ;
    public String getJobUser () ;
    public String getJobNumber () ;
}
```

The JSMClient object allows access to information from the client program.

6.2.5 JSMResource Interface

```
public interface JSMResource
{
    public Object getObject ( String key ) ;

    public String getResource ( String key ) ;

    public Enumeration getResourceNames () ;
}
```

The JSMResource object is a hash table of resource properties and values. The getObject method is used to access any non String value object. The key is always trimmed and uppercase.

6.2.6 JSMSStorage Interface

```
public interface JSMSStorage
{
    public void clear () ;
    public void remove ( String key ) ;

    public Object getObject ( String key ) ;
    public boolean putObject ( String key, Object value ) ;

    public boolean containsKey ( String key ) ;
}
```

The JSMSStorage object is a hash table that allows object to be stored and retrieved.

The key is always trimmed and uppercase.

6.2.7 JSMTrace Interface

```
public interface JSMTrace
{
    public void flush () ;

    public int getNumber () ;

    public File createTraceFile ( String fileName ) ;

    public void print ( String text ) ;
    public void println ( String text ) ;

    public void println ( Object object1, Object object2 ) ;
    public void println ( Object object1, Object object2, Object object3 ) ;
    public void println ( Object object1, Object object2, Object object3, Object object4 ) ;
    public void println ( Object object1, Object object2, Object object3, Object object4, Object object5 ) ;
    public void println ( Object object1, Object object2, Object object3, Object object4, Object object5, Object object6 ) ;
    public void println ( Object object1, Object object2, Object object3, Object object4, Object object5, Object object6, Object object7 ) ;
    public void print ( Throwable throwable ) ;
    public void print ( JSMCommand command ) ;
    public void print ( JSMCommand command, JSMResponse response ) ;
    public void print ( JSMCommand command, Throwable throwable ) ;
}
```

The JSMTrace object allows the programmer who wrote the service class to write out trace information.

If the JSMTrace object is not null, you can write to a trace file because it exists.

String text is written to the trace file UTF-8 encoded.

The println method appends CRLF (0x0D0x0A) to the end of the UTF-8 encoded bytes.

It is recommended not to embed "\n" in the String text, but use the print and println methods.

Example

```
public final void service ( JSMContainer container )
{
    m_trace = container.getServiceTrace () ;
}
```

```
public final JSMResponse command ( JSMCommand command ) throws JS
{
    try
    {
        if ( m_trace != null )
        {
            m_trace.print ( command ) ;
        }

        JSMResponse response = runCommand ( command ) ;

        if ( m_trace != null )
        {
            m_trace.print ( command, response ) ;
        }

        return response ;
    }
    catch ( Throwable t )
    {
        if ( m_trace != null )
        {
            m_trace.print ( command, t ) ;
        }

        return new JSMResponse ( t ) ;
    }
}
```

```
private final
JSMResponse runCommand ( JSMCommand command ) throws Exception
{
```

```
if ( command.equals ( command.SERVICE_LOAD ) )
{
    return new JSMResponse ( "Command has completed" );
}

return new JSMResponse ( JSMResponse.ERROR, "Unknown command"
}
```

6.2.8 JSMList Interface

If the optional working list argument on the JSM_COMMAND BIF is used, then the contents of this working list can be accessed using this interface.

```
public interface JSMList
{
    public int getFieldCount () ;
    public int getEntryCount () ;
    public int getMaxEntryCount () ;

    public Enumeration getNames () ;
    public Enumeration getLongNames () ;

    public String[] getNameArray () ;
    public String[] getLongNameArray () ;

    public Enumeration getFields () ;
    public JSMField[] getFieldArray () ;

    public int getIndex ( String name ) ;
    public boolean contains ( String name ) ;

    public String[] newEntryArray () ;
    public String[] getEntryArray ( int index ) ;

    public void clear () ;
    public void remove ( int index ) ;
    public void add ( String[] data ) ;
    public void set ( int index, String[] data ) ;
    public void insert ( int index, String[] data ) ;
}
```

6.2.9 JSMFieldList Interface

If the SERVICE_EXCHANGE(*FIELD) keyword or the optional working list argument on the JSM_COMMAND BIF is used, then function fields can be accessed using this interface.

```
public interface JSMFieldList
{
    public int getFieldCount () ;

    public Enumeration getNames () ;
    public Enumeration getLongNames () ;

    public String[] getNameArray () ;
    public String[] getLongNameArray () ;

    public Enumeration getFields () ;
    public JSMField[] getFieldArray () ;

    public boolean contains ( String name ) ;

    public JSMField getField ( String name ) ;

    public String getValue ( String name ) ;
    public void setValue ( String name, String value ) ;
}
```

6.2.10 JSMField Interface

```
public interface JSMField
{
    public String getName ();
    public String getLongName ();

    public DataType getDataType ();
}
```

6.3 Java Service Manager Classes

The Java Service Manager is shipped with the following classes:

[6.3.1 JSMPResponse Class](#)

[6.3.2 JSMPException Class](#)

[6.3.3 Miscellaneous Helper Class](#)

[6.3.4 Native IBM i Classes](#)

[6.3.5 Utility Classes](#)

6.3.1 JSMResponse Class

The JSMReponse object is used to return a response to the JSM client program. The JSMResponse object contains a status, message, byte array, list and field list object.

The Unicode status string value will be encoded to the native encoding of the client program.

The status value is copied into the #JSMSTS field of the LANSAs Built-In Function API's.

The Unicode message string value will be encoded to the native encoding of the client program.

The message value is copied into the #JSMMSG field of the LANSAs Built-In Function API's.

For Single-byte environments, Unicode characters encode to one byte for each character.

For Double-byte environments, Unicode characters encode to one byte for each character if no DBCS characters are present.

If DBCS characters are present, then Shift-Out and Shift-In characters will mark the change from one byte representing a character to two bytes representing a character.

If the byte size of status and message exceed their limits, an exception will be thrown during the send response stage.

The byte array variable content is copied to memory allocated within the JSM client program. No conversion is performed on the data.

Each new JSM command call, frees the memory allocated to store the previous command's byte array response.

The byte array memory also contains the HTTP response that will be passed onto the HTTP server when a JSM_CLOSE is performed.

A JSM_CLOSE call also frees the memory allocated by a byte array response.

It is recommended to always call JSM_CLOSE before ending a JSM client program.

The internal default settings are:

Status OK

Message zero length string object

ByteArray zero length byte array object

List null object

Fieldlist null object

Two static final string variables called OK and ERROR are available for ease of use.

Example

```
new JSMResponse ( "Ok message" );  
new JSMResponse ( JSMResponse.OK, "Ok message" );  
new JSMResponse ( JSMResponse.ERROR, "Error message" );  
new JSMResponse ( exception );
```

6.3.1.1 Constructors

```
public JSMResponse ( String message )
public JSMResponse ( String status, String message )
public JSMResponse ( String status, String message, JSMList list )
public JSMResponse ( String status, String message, JSMFieldList fieldList )
public JSMResponse ( String status, String message, byte[] byteArray )

public JSMResponse ( JSMList list )
public JSMResponse ( JSMFieldList fieldList )
public JSMResponse ( JSMFieldList fieldList, JSMList list )

public JSMResponse ( byte[] byteArray )

public JSMResponse ( Throwable t )
```

If the constructor parameter is a Throwable object then the status value is ERROR and the message value comes from the Throwable object.

If the Throwable object is an instance of JSMEException then the status and message values come from the JSMEException object.

If the Throwable getMessage method returns a null, then the message value is the class name of the Throwable object.

6.3.1.2 Getter/Setter Methods

```
public final String getStatus ()  
public final void setStatus ( String status )
```

The status parameter value is trimmed and converted to uppercase.
If the status parameter is null, then the internal status value is set to ERROR.

```
public final String getMessage ()  
public final void setMessage ( String message )
```

If the message parameter is null, then the internal message value is set to a zero length string.

```
public final JSMList getList ()  
public final void setList ( JSMList list )
```

```
public final JSMFieldList getFieldList ()  
public final void setFieldList ( JSMFieldList fieldList )
```

```
public final byte[] getByteArray ()  
public final void setByteArray ( byte[] byteArray )
```

If the byteArray parameter is null, then the internal byteArray value is a zero length byte array object.

6.3.2 JSMEException Class

The JSMEException object is used to return a response to the JSM client program.

The JSMEException object contains a status object. The message object is stored in the base Exception class.

The Unicode status string value will be encoded into the native encoding of the client program. This status string value is copied into the #JSMSTS field of the LANSA BIF API's.

The status value cannot exceed 20 bytes.

The Unicode message string value will be encoded into the native encoding of the client program. This message string value is copied into the #JSMMSG field of the LANSA BIF API's.

The message value cannot exceed 255 bytes for RDML and RPG clients.

The internal default settings are:

status ERROR

6.3.2.1 Constructors

```
public JSMEException ( Throwable t )  
public JSMEException ( String message )  
public JSMEException ( String status, String message )
```

The main purpose of JSMEException is to allow a response to be returned to the client program when service program code is several layers down and it is not easy to return a JSMPResponse object.

If the constructor parameter is a Throwable object then the status value is ERROR and the message value comes from the Throwable object.

If the Throwable getMessage method returns a null, then the message value is the class name of the Throwable object.

Throwing a JSMEException or returning a JSMPResponse object are the two ways of returning data and control to the client program.

The default status is ERROR.

6.3.2.2 Other Exception Handling

An exception not handled by the service program code will be caught and a response sent back to the client program.

The status field value of these responses is always the string FATAL.

The message field value is useful information explaining the exception, such as the exception message.

Possible status field values:

- OK The OK status is the default for the JSMResponse class.
- ERROR The ERROR status is the default for the JSMEException class.
- FATAL An exception has occurred.
- User-
defined The JSMResponse and JSMEException status field can be set by the
 service program code.

6.3.3 Miscellaneous Helper Class

The static method `JSMManager.getTemporaryDirectory` returns the temporary directory of the current service as a `java.io.File` object.

The temporary directory is not created by this method.

The name of this temporary directory is the job number and hash code of the service thread.

Example

```
File directory = JSMManager.getTemporaryDirectory () ;
```

The static method `JSMManager.createTemporaryDirectory` returns the temporary directory of the current service as a `java.io.File` object.

The temporary directory is created by this method.

The name of this temporary directory is the job number and hash code of the service thread.

Example

```
File directory = JSMManager.createTemporaryDirectory () ;
```

The static method `JSMManager.createTemporaryFile` returns the temporary file as a `java.io.File` object.

If the temporary directory does not exist then it is created, but the temporary file is not created.

The *filename* only needs to be unique within the service program, so a simple name like "upload" can be used.

To read and write to the file use the `FileInputStream` and `FileOutputStream` classes.

Example

```
File file = JSMManager.createTemporaryFile ( filename ) ;
```

The temporary directory only exists between a JSM OPEN and JSM CLOSE.

The temporary directory is cleared on the JSM OPEN.

The temporary directory is also cleared on the JSMCLOSE.

There is no need for the programmer to clear the temporary directory.

The static method JSManager.clearTemporaryDirectory deletes all files in the temporary directory and then deletes the temporary directory.

Example

```
boolean ok = JSManager.clearTemporaryDirectory () ;
```

When a service thread starts a call to the JSManager.clearTemporaryDirectory method is performed to make sure no previous temporary files from a previous thread with the same hash code exist.

The CLRJSM command can optionally remove all files and directories beneath the temp and trace directory.

6.3.4 Native IBM i Classes

Several classes have been developed to give access directly to IBM i data objects.

These classes are separate from JSM and are located in the jsmnative.jar file. The Data Queue, Message Queue and User Space classes use IBM i service programs.

6.3.4.1 DataQueue Class

The DataQueue class allows a service program to clear, send and receive messages from a data queue.

This class uses the native IBM i service program JSMDTAQ.

6.3.4.2 MessageQueue Class

The MessageQueue class allows a service program to send information text messages to message queues.

This class uses the native IBM i service program JSMMSGQ.

6.3.4.3 UserSpace Class

The UserSpace class allows a service program to create, read and write to user spaces.

This class uses the native IBM i service program JSMUSRSPCE.

6.3.4.1 DataQueue Class

Constructors

DataQueue (String library, String queue)

Methods

String getEncoding ()

String check ()

String check (String authority)

int getKeyLength ()

int getMessageLength ()

boolean clear ()

boolean clear (String key)

boolean write (String data)

boolean write (byte[] data)

boolean write (String key, String data)

boolean write (String key, byte[] data)

byte[] read (int waitTime)

byte[] read (String key, int waitTime)

String readString (int waitTime)

String readString (String key, int waitTime)

Maximum sizes

Queue length	10 bytes
Library length	10 bytes
Authority length	10 bytes
Max Key length	256 bytes

Max Message length	64512 bytes
--------------------	-------------

Comments

The write method will convert the Unicode String into bytes using the CCSID of the job.

When the JNI service writes the byte message to the data queue, the message is padded with white space.

The read method will read the next message from the data queue.

The readString method will read the next message from the data queue and convert it to a Unicode string.

The data queue wait time on the read method only takes effect if it is the first thread to attempt to read data.

The check method returns one of the following values.

Null	Internal Error
Zero length string	OK
CPFnnnn	i5/OS message Id

6.3.4.2 MessageQueue Class

Constructors

No constructors

Static Methods

MessageQueue queueSystemOperator ()

MessageQueue queueActiveUsers ()

MessageQueue queueHistoryLog ()

MessageQueue queueUser (String user)

MessageQueue queueSpecified (String queue, String library)

Methods

String getEncoding ()

boolean send (String message)

Maximum Sizes

Queue length 10 bytes

Library length 10 bytes

Message length 6000 bytes

Comments

The send method will convert the Unicode String into bytes using the CCSID of the job.

6.3.4.3 UserSpace Class

Constructors

UserSpace (String library, String name)

Methods

String getEncoding ()

String check ()

String check (String authority)

boolean create ()

boolean create (int initialSize)

boolean create (int initialSize, String description)

boolean create (int initialSize, String authority, String description)

boolean delete ()

boolean write (String data)

boolean write (byte[] data)

boolean write (int startPosition, String data)

boolean write (int startPosition, byte[] data)

byte[] read (int readLength)

byte[] read (int startPosition, int readLength)

String readString (int readLength)

String readString (int startPosition, int readLength)

Maximum sizes

Library length	10 bytes
Name length	10 bytes
Authority length	10 bytes
Description length	50 bytes

Max user space size	16776704 bytes
---------------------	----------------

Defaults

Default user space size	8192 bytes
Default description	"Created by Java Service Manager (JSM)"
Default create authority	*CHANGE
Default check authority	*USE
Default start position	1

Public static constants

AUTHORITY_ALL	*ALL
AUTHORITY_USE	*USE
AUTHORITY_EXCLUDE	*EXCLUDE
AUTHORITY_CHANGE	*CHANGE
AUTHORITY_LIBCRTAUT	*LIBCRTAUT

Comments

The check method has the same functionality as the DataQueue class.

The write method will convert the Unicode String into bytes using the CCSID of the job.

The read method will read the specified number of bytes from the user space.

The readString method will read the specified number of bytes from the user space and return it as a Unicode string.

6.3.5 Utility Classes

Several classes have been developed to allow handling of IBM i data types.

DataType

All `DataType` classes implement the `DataType` interface.

```
public int getType ()  
public int getByteLength ()  
public byte[] toByteArray ( Object data )  
public String toString ( byte[] byteArray )
```

6.3.5.1 DataTypeText

This class allows a Java data type to be converted to an alphanumeric byte array.

6.3.5.2 DataTypePacked

This class allows a Java data type to be converted to a packed decimal byte array.

6.3.5.3 DataTypeZoned

This class allows a Java data type to be converted to a zoned decimal byte array.

6.3.5.1 DataTypeText

Constructors

```
DataTypeText ( int length, String encoding )
```

Methods

```
int getType ()  
int getByteLength ()  
String getEncoding ()  
byte[] toByteArray ( Object data )  
String toString ( byte[] byteArray )
```

Example

```
DataType dataType = new DataTypeText ( 50, encoding )  
byte[] byteArray = dataType.toByteArray ( "A quick brown fox" );
```

6.3.5.2 DataTypePacked

Constructors

```
DataTypePacked ( int digits )  
DataTypePacked ( int digits, int scale )
```

Methods

```
int getType ()  
int getByteLength ()  
int getDigits ()  
int getScale ()  
byte[] toByteArray ( Object data )  
String toString ( byte[] byteArray )
```

Example

```
DataType dataType = new DataTypePacked ( 10, 2 )  
byte[] byteArray = dataType.toByteArray ( "240.00" );
```

6.3.5.3 DataTypeZoned

Constructors

```
DataTypeZoned ( int digits )  
DataTypeZoned ( int digits, int scale )
```

Methods

```
int getType ()  
int getByteLength ()  
int getDigits ()  
int getScale ()  
byte[] toByteArray ( Object data )  
String toString ( byte[] byteArray )
```

Example

```
DataType dataType = new DataTypeZoned ( 10, 2 )  
byte[] byteArray = dataType.toByteArray ( "240.00" );
```

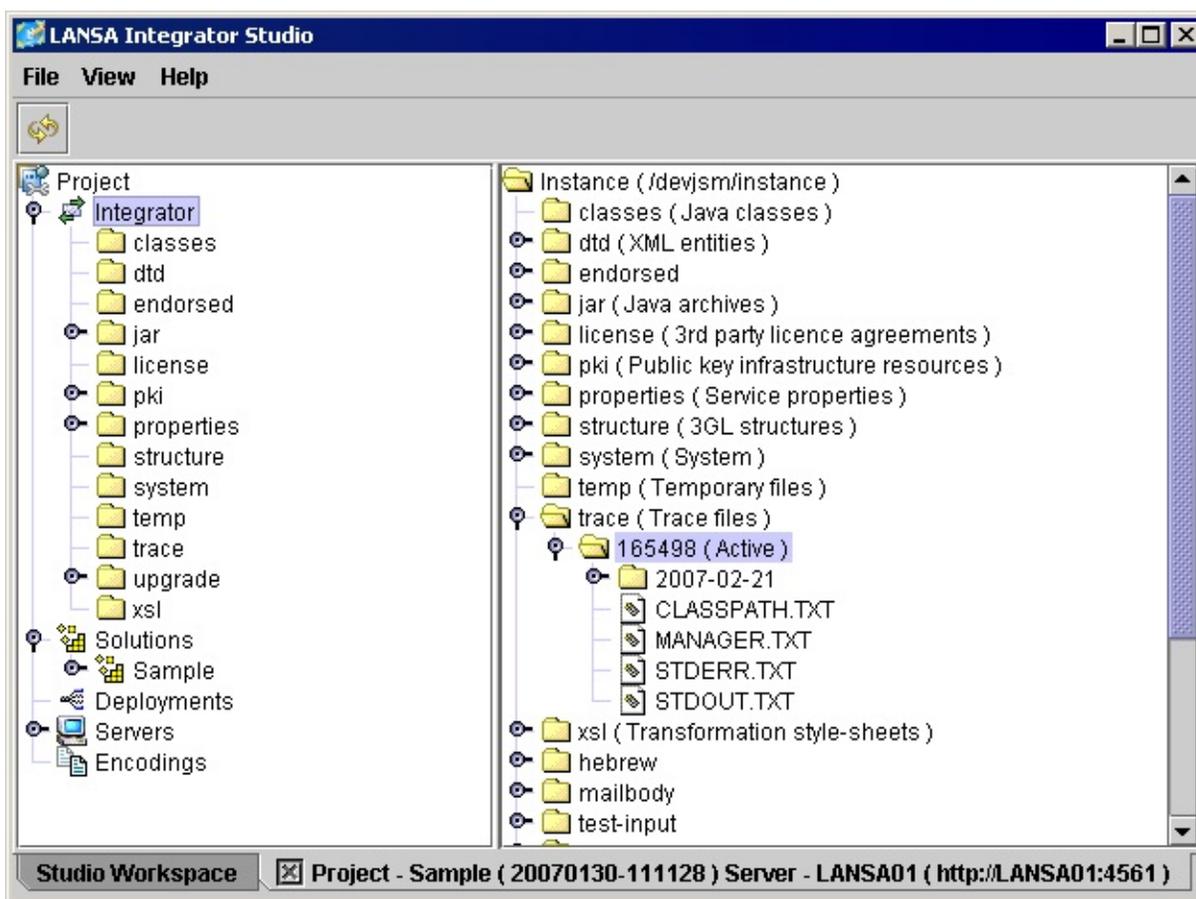
7. Integrator Studio

Integrator Studio is an integrated desktop application that allows a single point of management for JSM Server instances as well as the resources used by your Integrator applications or projects.

Studio's easy to use graphical interface greatly simplifies the configuration and setup of the JSM server environment. For example, developers can display instance information, update JSM configuration files on the server, download trace files, and publish the generated XSL files to the JSM server.

Using Studio, files can be published to or retrieved from the instance server using simple drag and drop operations or by using context (pop-up) menus.

Integrator Studio also provides a single point of access for the Integrator tools.



7.1 Get Started with Studio

Starting Studio

To access Integrator Studio, click on the Integrator Studio program icon in the LANSAs folder on your desktop. This icon is installed with the Studio software. The [7.1.1 Studio Main Window](#) will open.

Studio Program Short Cut

The program short cut uses the LANSALauncher.exe program, which uses javaw.exe to execute the Studio application. The command line for LANSALauncher.exe identifies the short cut to be used for Studio and should not be modified.

```
"C:\Program Files\LANSAsxxx\Tools\LANSALauncher.exe" /app=javaw.exe /i
```

The actual command line arguments for javaw.exe are in the LANSALauncher.cfg file, located in the same folder as the LANSALauncher.exe program.

```
[id:Studio]
arg=-Djava.ext.dirs="<JRE_LIB_EXT>;.\lib\ext"
  -Djava.endorsed.dirs=".\lib\endorsed" com.lansa.jsf.studio.JSFStudio
workdir=C:\PROGRA~1\LANSAsxxx\INTEGR~1\Studio
```

The LANSALauncher.cfg file can be edited using a text editor.

The arg entry under [id:Studio] can be modified to change the Java environment.

If the memory requirement of the JVM needs to be increased, then the -Xms and -Xmx parameters need to be added to the arg entry.

```
[id:Studio]
arg=-Xms256m -Xmx256m
  -Djava.ext.dirs="<JRE_LIB_EXT>;.\lib\ext"
  -Djava.endorsed.dirs=".\lib\endorsed" com.lansa.jsf.studio.JSFStudio
workdir=C:\PROGRA~1\LANSAsxxx\INTEGR~1\Studio
```

The JRE lib\ext directory contains additional Java jar files.

JCA/JCE providers for Java Cryptography APIs

sunjce_provider.jar
sunec.jar
sunpkcs11.jar
sunmscapi.jar

JNDI DNS provider

dnsns.jar

Non US English locale resources

localedata.jar

Shared Studio Projects

To allow users to access shared projects, specify the shared directory as a Studio program argument. Modify an existing [id:Studio] section in the **LANSALauncher.cfg** file and add the shared directory to the end of the **arg** entry or copy an existing Studio short cut, change the **id** to Studio2, copy the existing [id:Studio] entries, paste them, rename the new section to [id:Studio2] and add the shared directory to the end of the program arg entry under the [id:Studio2] section.

com.lansa.jsf.studio.JSFStudio "s:\studio"

A workspace sub directory and Studio.xml file will be created in the specified directory.

A temp sub directory used for Drag and Drop operations will be created in the program's current or start in directory on the local drive.

By default Studio uses the current directory or start in directory as its working directory.

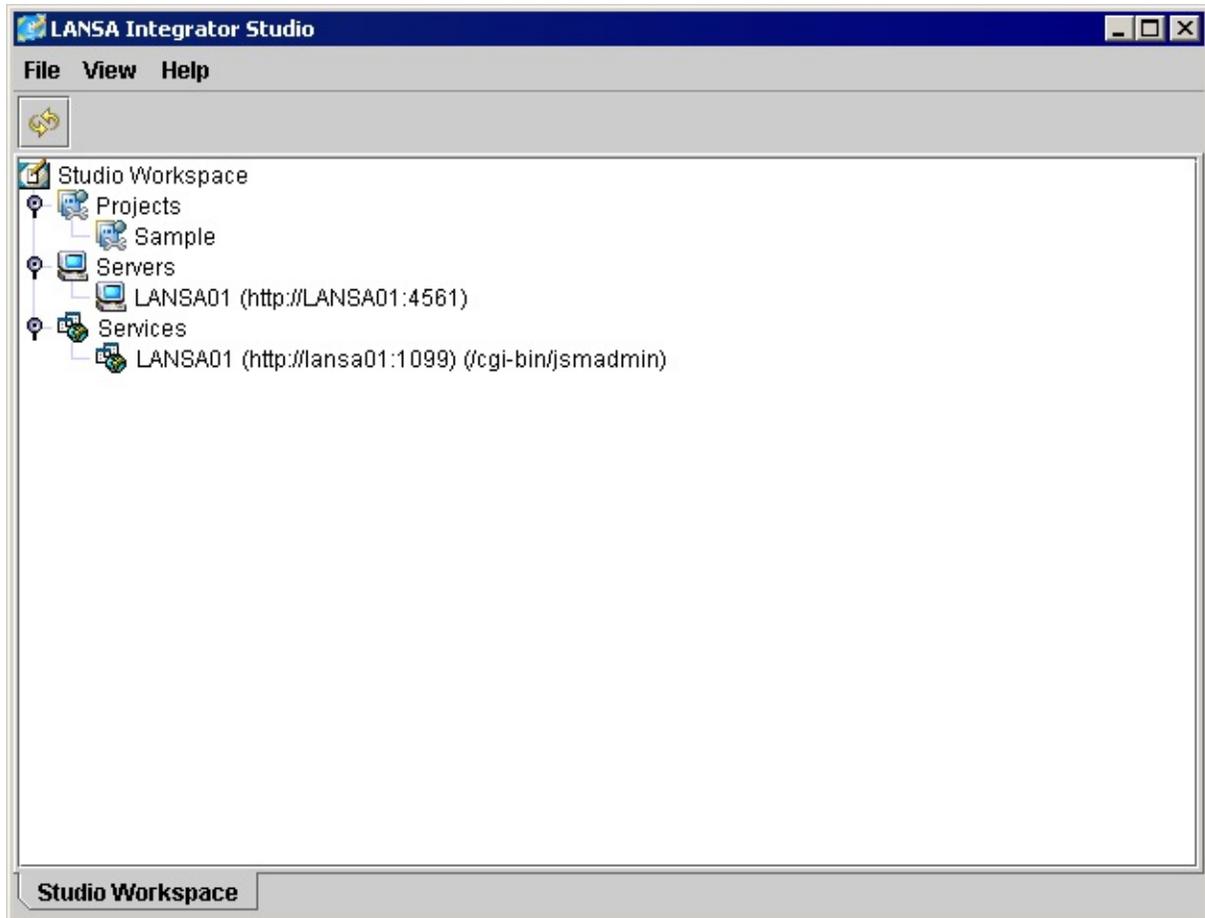
Prerequisites

Before you begin to use Integrator Studio, check that you have the following:

- Do you have a TCP/IP connection to the JSM server and HTTP server?
- If you are accessing a JSM Instance, has the JSM server been started? To communicate with the JSM server, the JSM must be running. The instance's `manager.properties` file should also contain entries for `studio.client.address` and possibly `studio.authentication` (see [Java Service Manager Console](#)).

7.1.1 Studio Main Window

Once opened, the Studio main window will appear something like this:



The Studio Workspace is a tree containing three main sections:

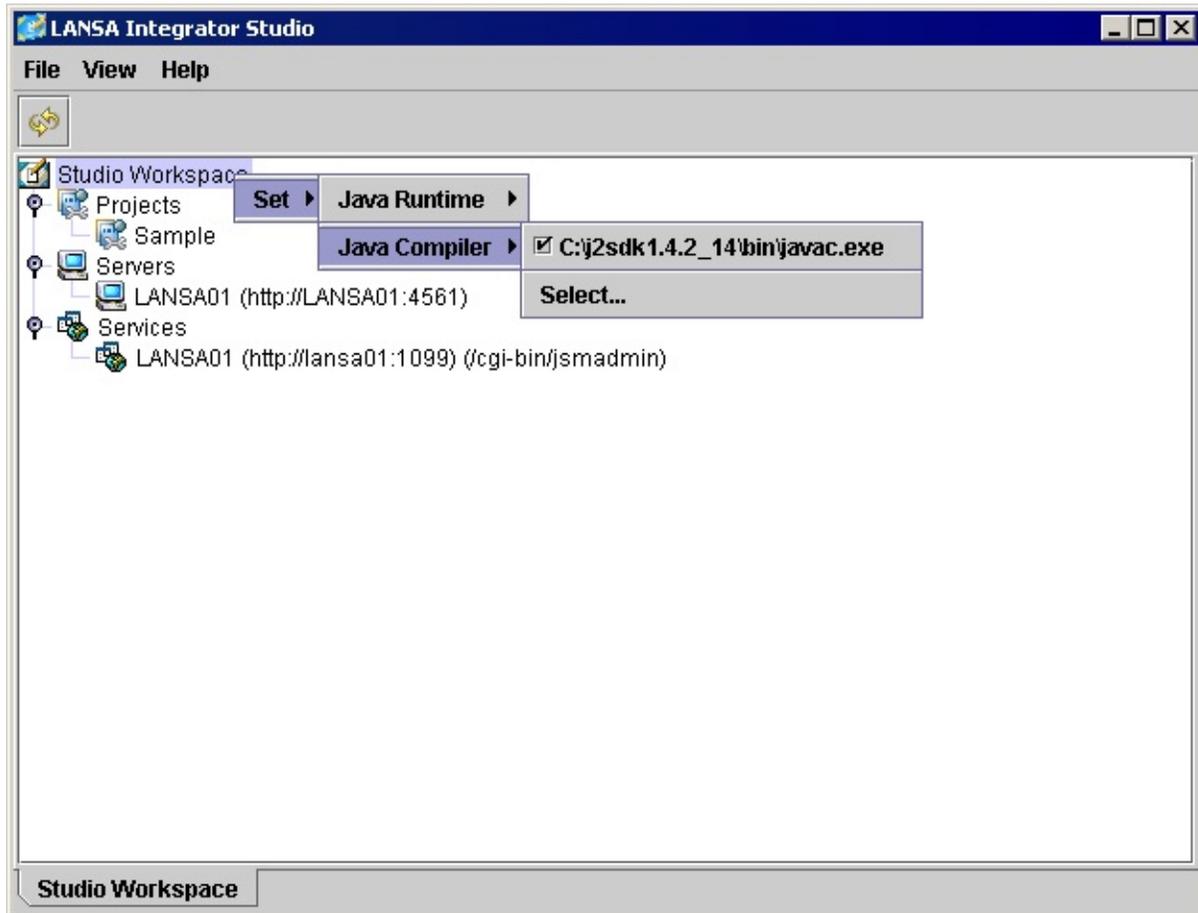
- Projects
- Servers
- Services

Use the context (pop-up) menu available on each of these nodes to create a new Project, Server or Service.

Use the context menu available on each child node to open, rename, delete or to perform some other task associated with the selected node.

7.1.2 Set Java Compiler and Java Runtime

Wizards that generate and compile Java code require a Java compiler, such as Oracle's Java SDK, to be installed on the PC. Use the Set option to specify the location of the javac and java programs.



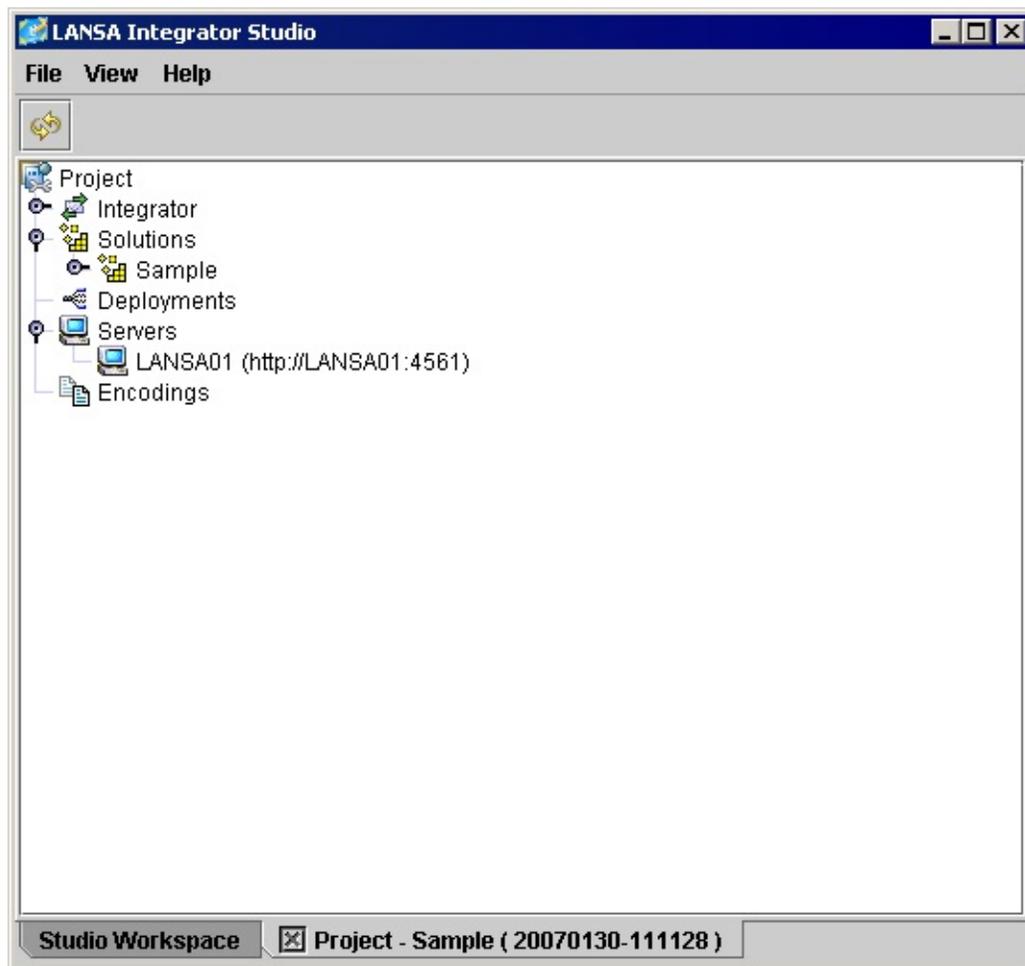
7.2 Use Studio with Projects

A Studio *project* is a developer's collection of files related to a particular application or task using LANSAs Integrator. You will create projects to help you manage the Integrator development environment. A project is like a working space where you can organize and store your files.

Projects are also very important when configuring properties for the JSM server. A Project will uniquely tag a section of a properties file to store settings specific to your project. You can retrieve and publish settings specific to a project rather than editing the whole properties file.

The directory structure of a project reflects the structure of a JSM Instance, so that files can be organized and easily transferred to or from the instance.

A project can be associated with one or more JSM servers.

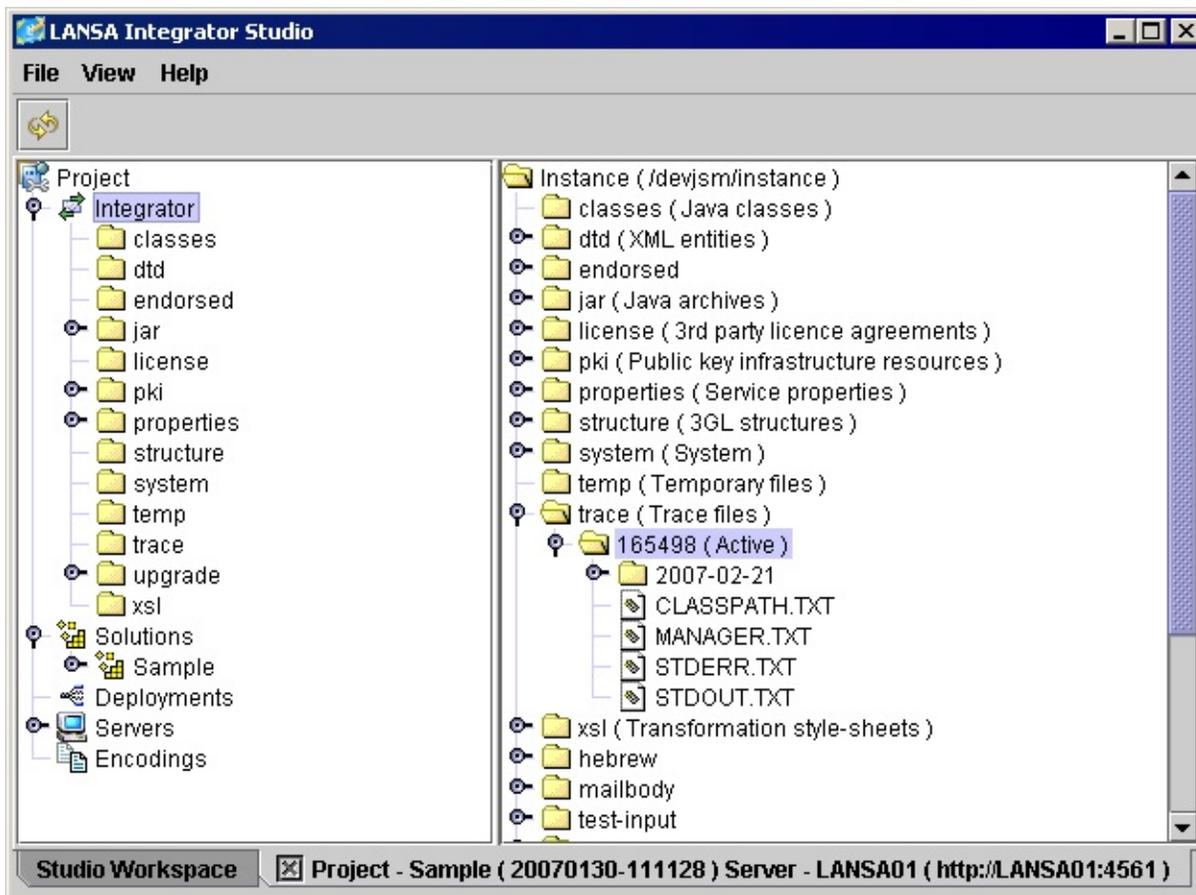


7.3 Use Studio with Servers

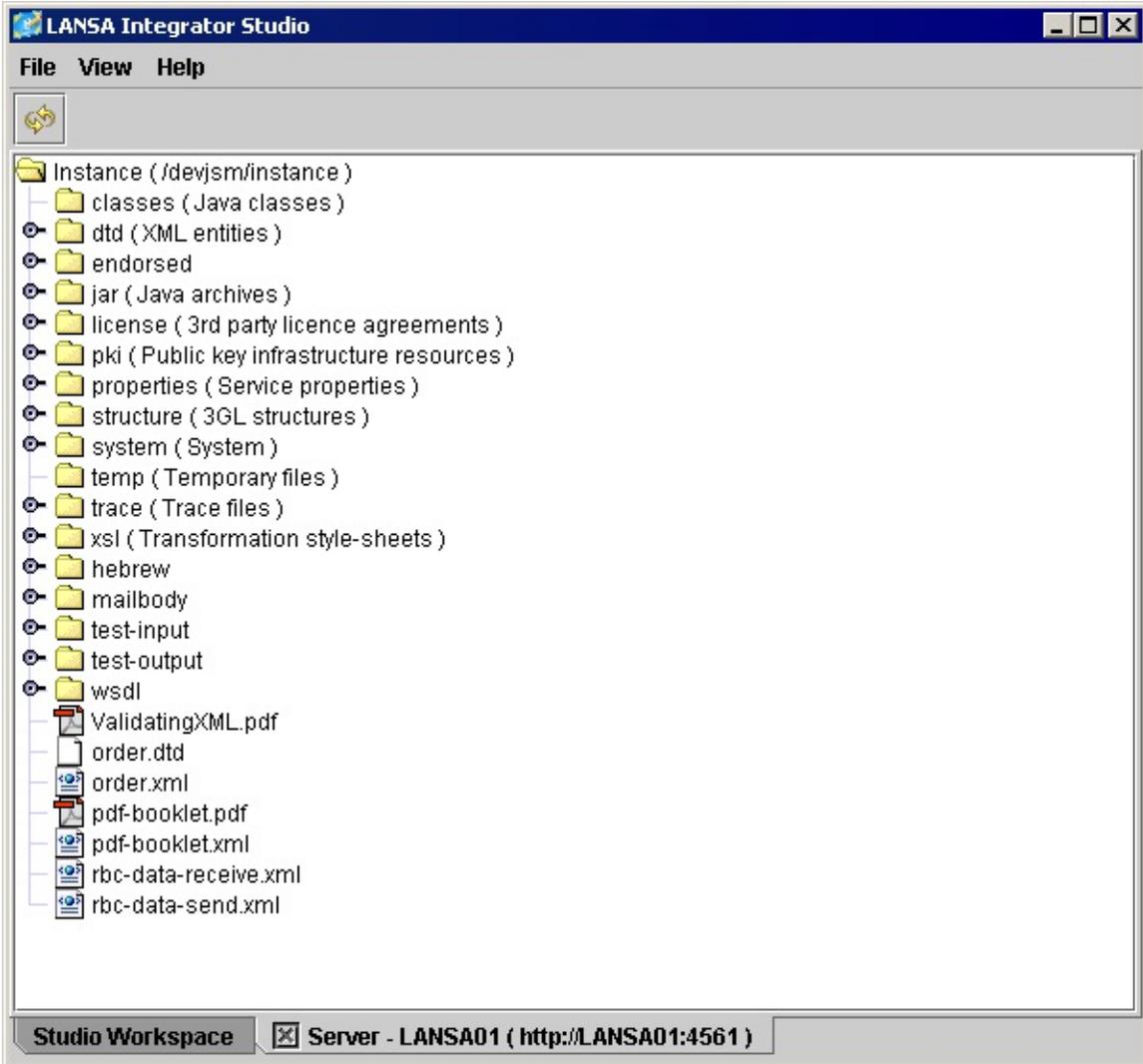
Studio allows you to view the contents of the instance directory of a JSM server. Using Studio, you can transfer the JSM instance files to and from the local machine.

When using Studio to work with JSM servers, you have two choices:

1. Access the JSM server using a project.

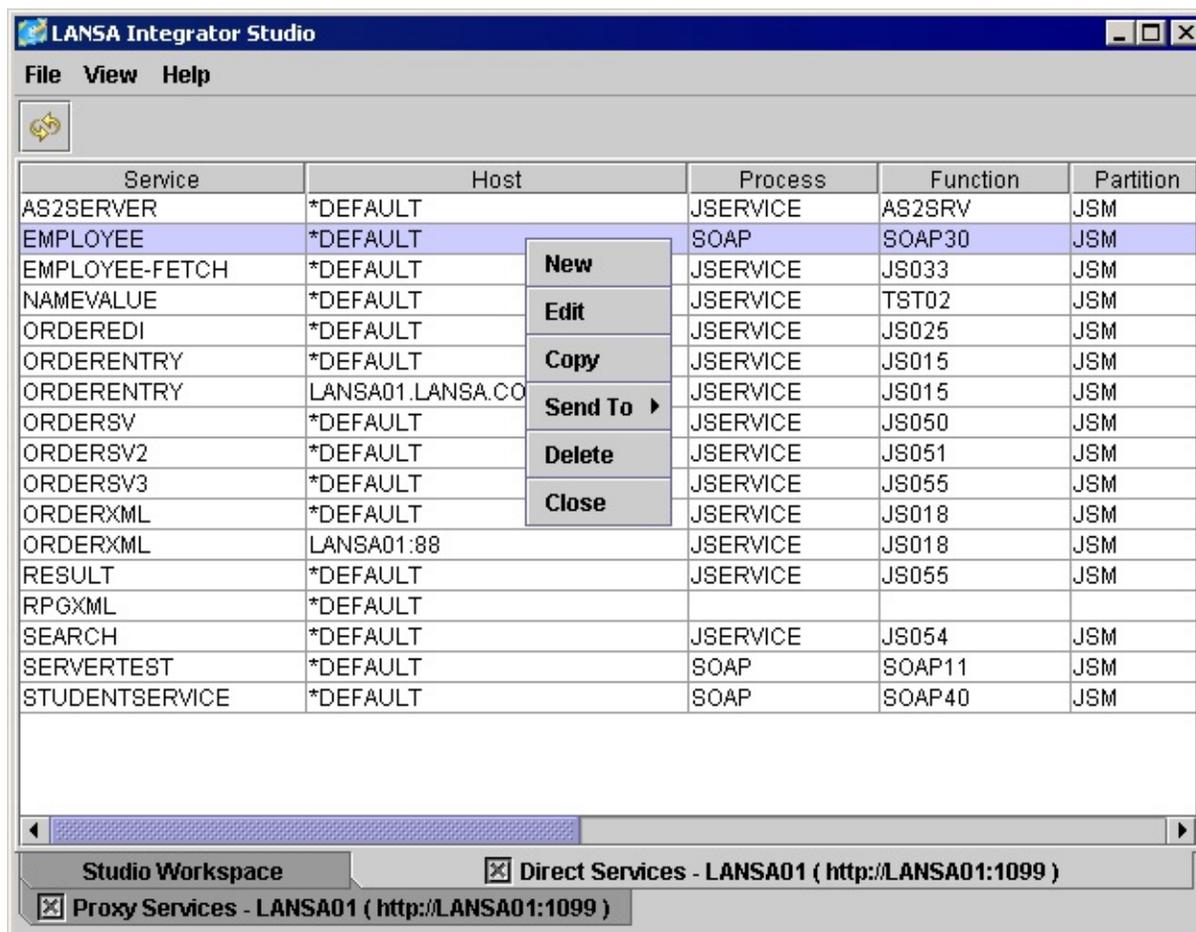


2. Access the JSM server directly.



7.4 Use Studio with Services

Studio allows you to maintain the contents of the JSMDirect and JSMProxy databases (DC@W29 and DC@W30 on IBM i; dc_w29.txt and dc_w30.txt on Linux and Windows).



Click the right mouse button on any entry in the Direct Services or Proxy Services tab to display the pop-up menu.

If the database table is empty, then right click on the table column to display the pop-up menu with a *New* menu item

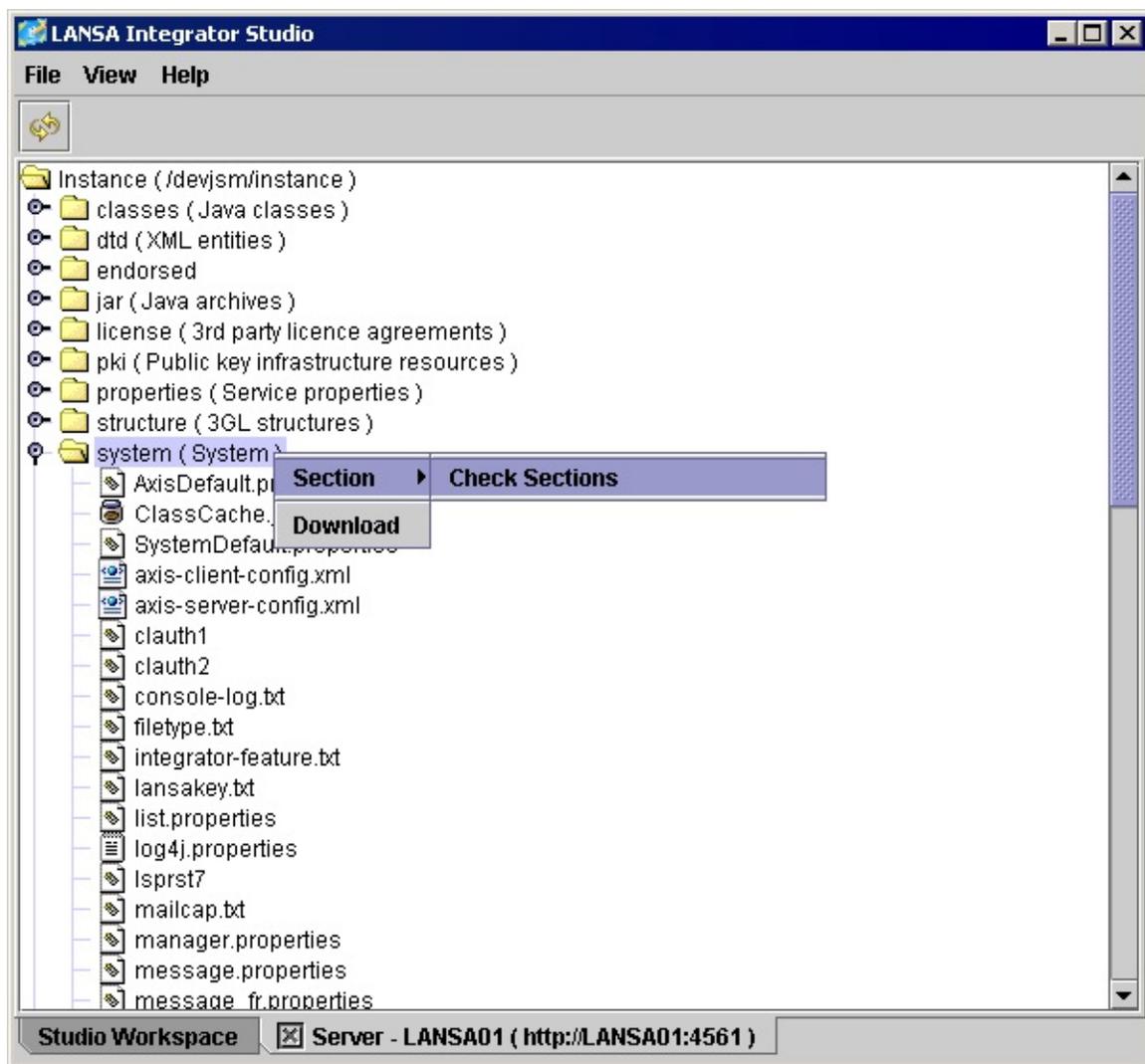
7.5 Use Studio to check sections

Studio allows you to maintain the Studio sections in the system and service property files.

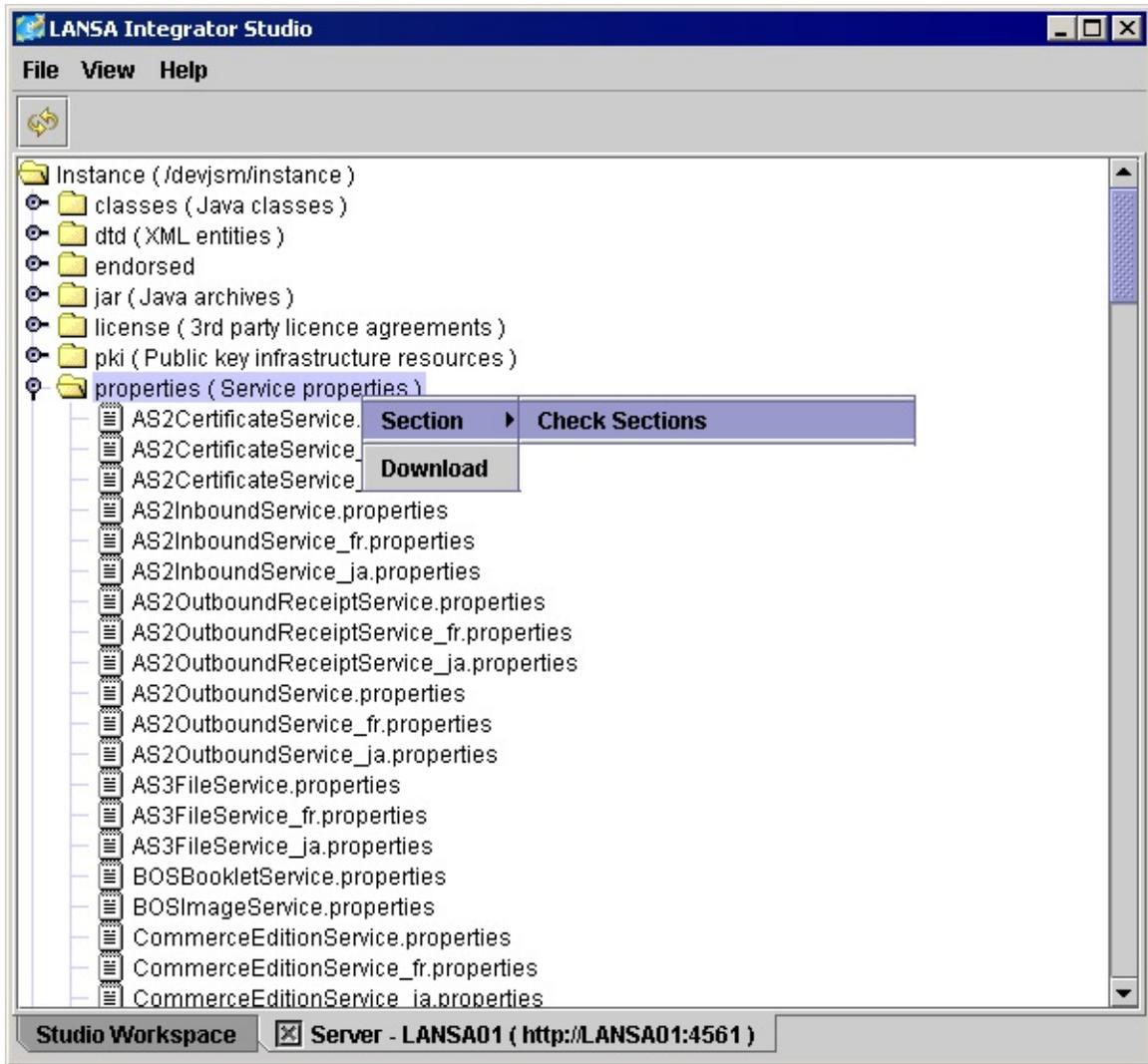
These Studio property files are located in the system and properties directory. Studio can check and remove unassigned sections at a directory or individual file level.

For more information about sections refer to [7.6.3 Publish and Retrieve](#).

To check all property files in the system directory, use the popup menu available from the directory node.



To check all property files in the properties directory, use the popup menu available from the directory node.



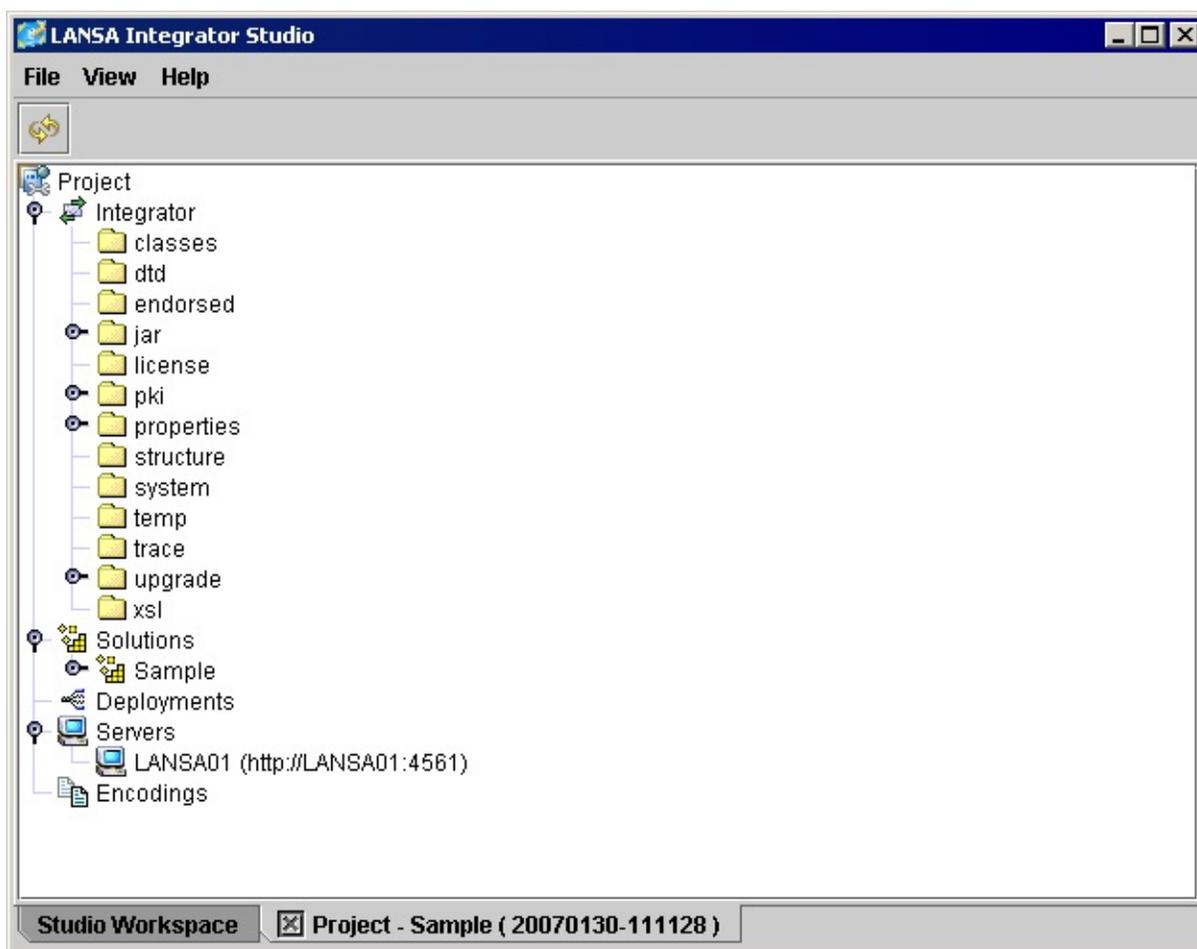
To check an individual file, use the popup menu available from the file node.

7.6 Projects

A Studio *project* is a collection of files related to a particular task or application using LANSAs Integrator. The directory structure of a project reflects the structure of a JSM Instance, so that files can be organized and published to the instance easily. A single project can be associated with more than one JSM Instance. A JSM Instance can be used by many different projects.

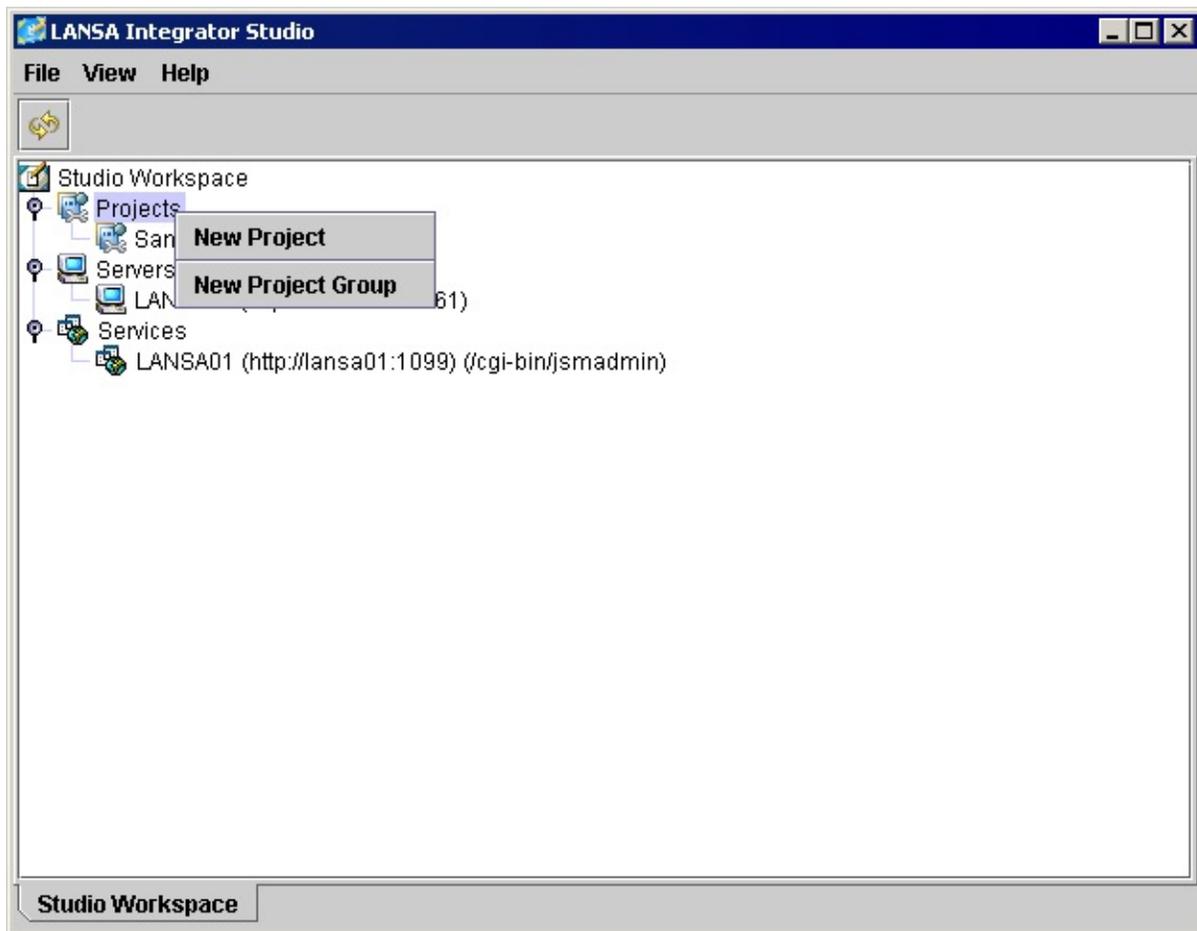
Projects are fully integrated with all the Studio Tools. When a tool is launched from Studio, it will place any files it creates in the appropriate place in the current Studio project.

Projects also allow you to uniquely tag configuration settings within a properties file used by a JSM Instance in order to simplify setup tasks.



7.6.1 New Project

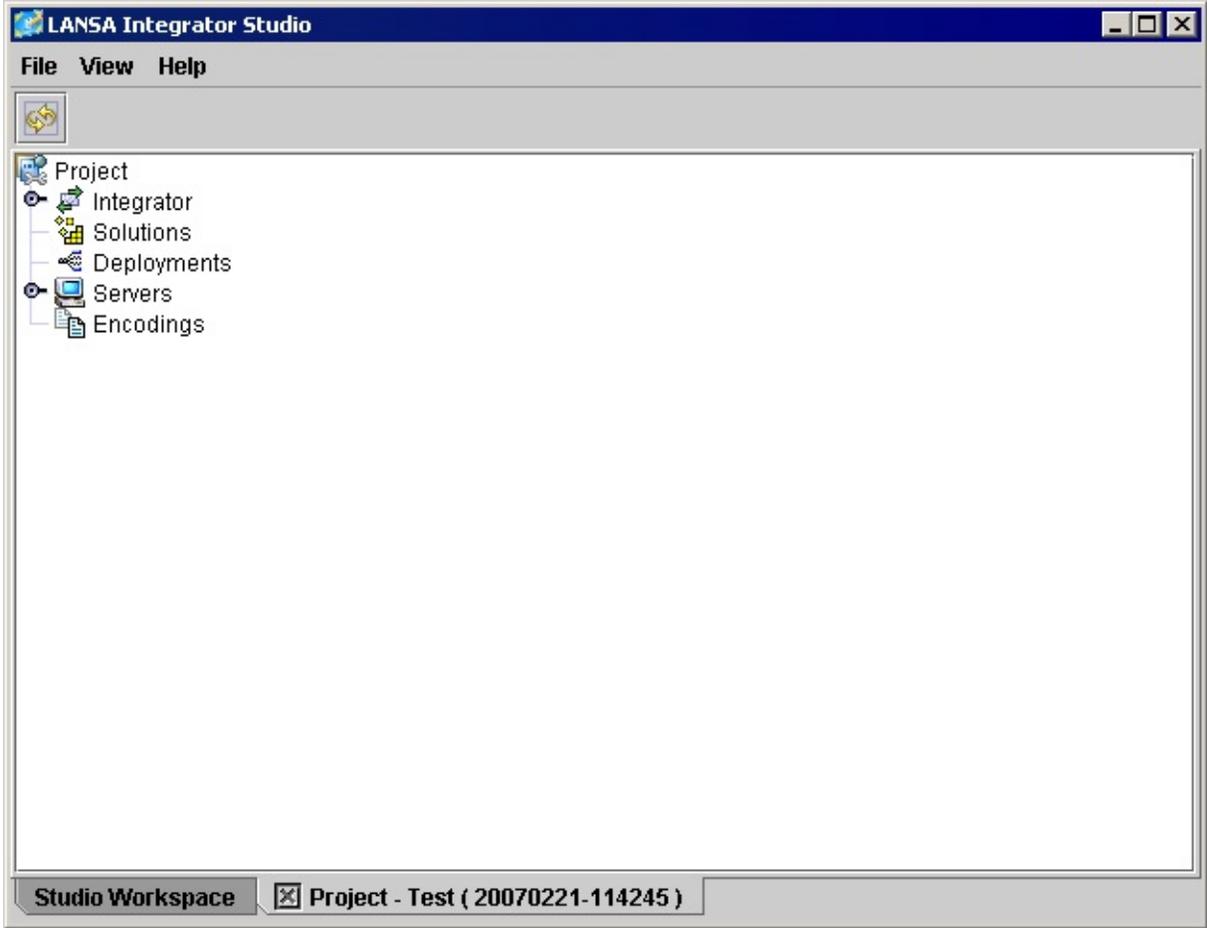
To create a new project, right click the Projects node in the Studio Workspace panel.



Enter the new project name using the prompt dialog.

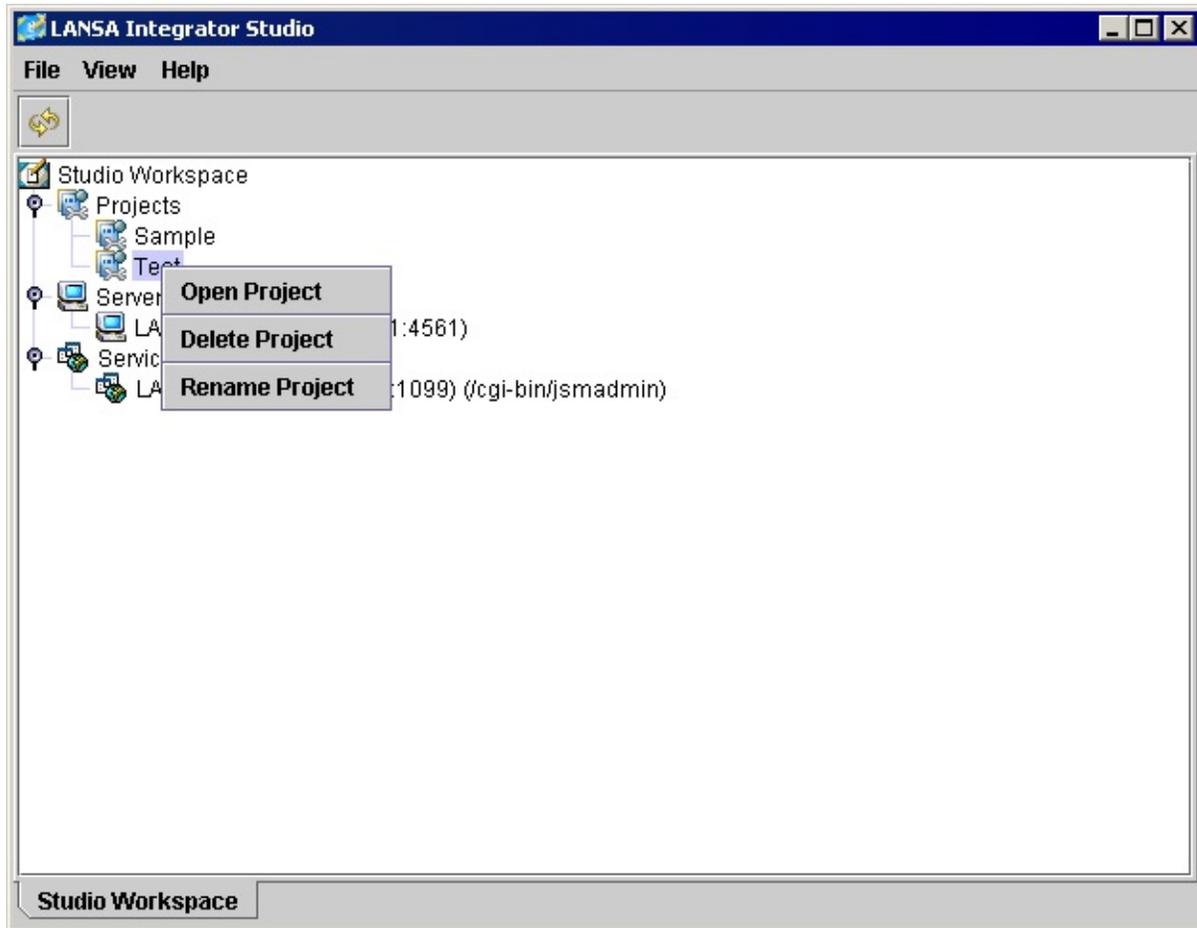


The Test project will be created and opened and this project will become the selected tab. A project Id is created using the System date and time. This project Id is used to mark Studio sections in the JSM properties files and is displayed after the project name on the tab label.

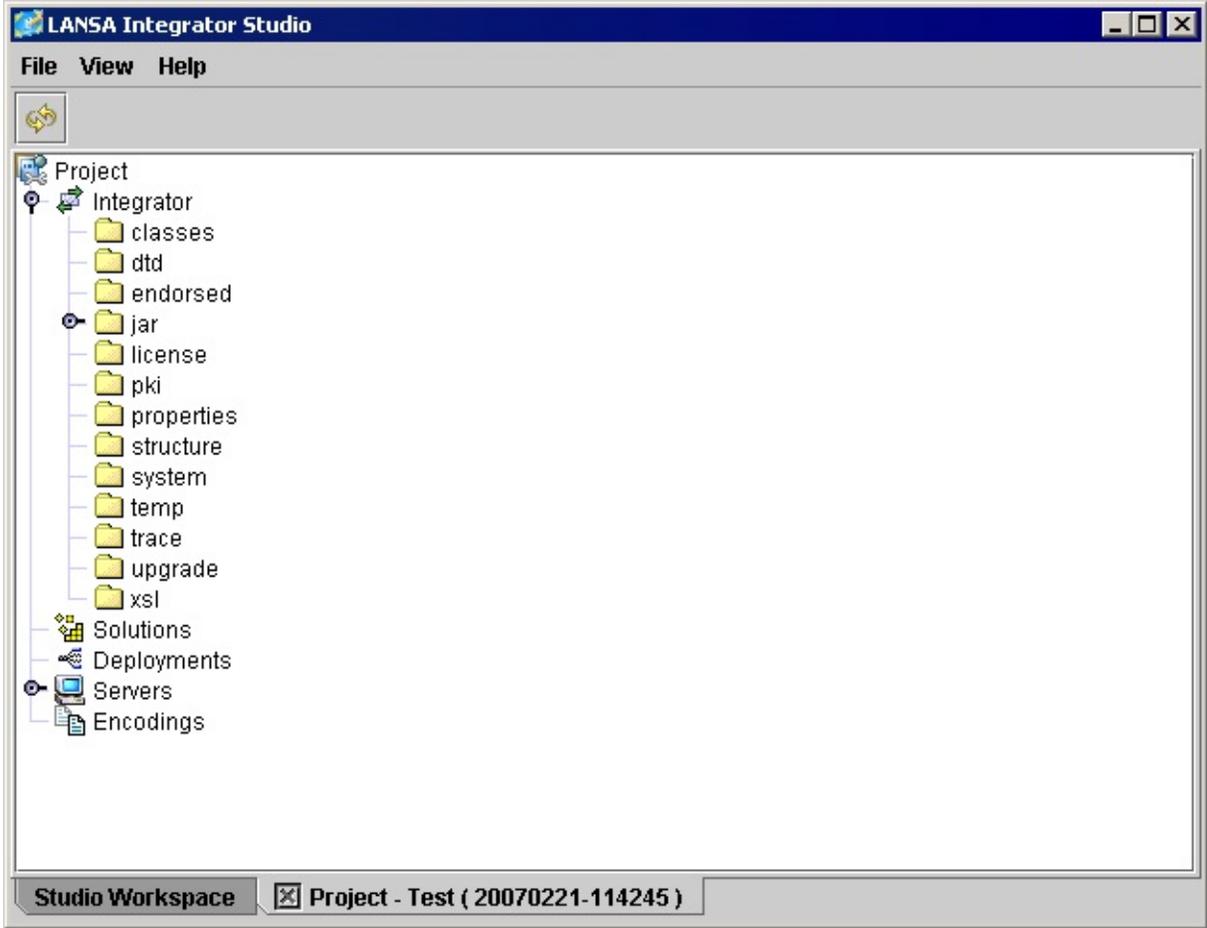


7.6.2 Open Project

From the Studio Workspace panel select the project to open and double-click or use the pop-up menu to open the project.



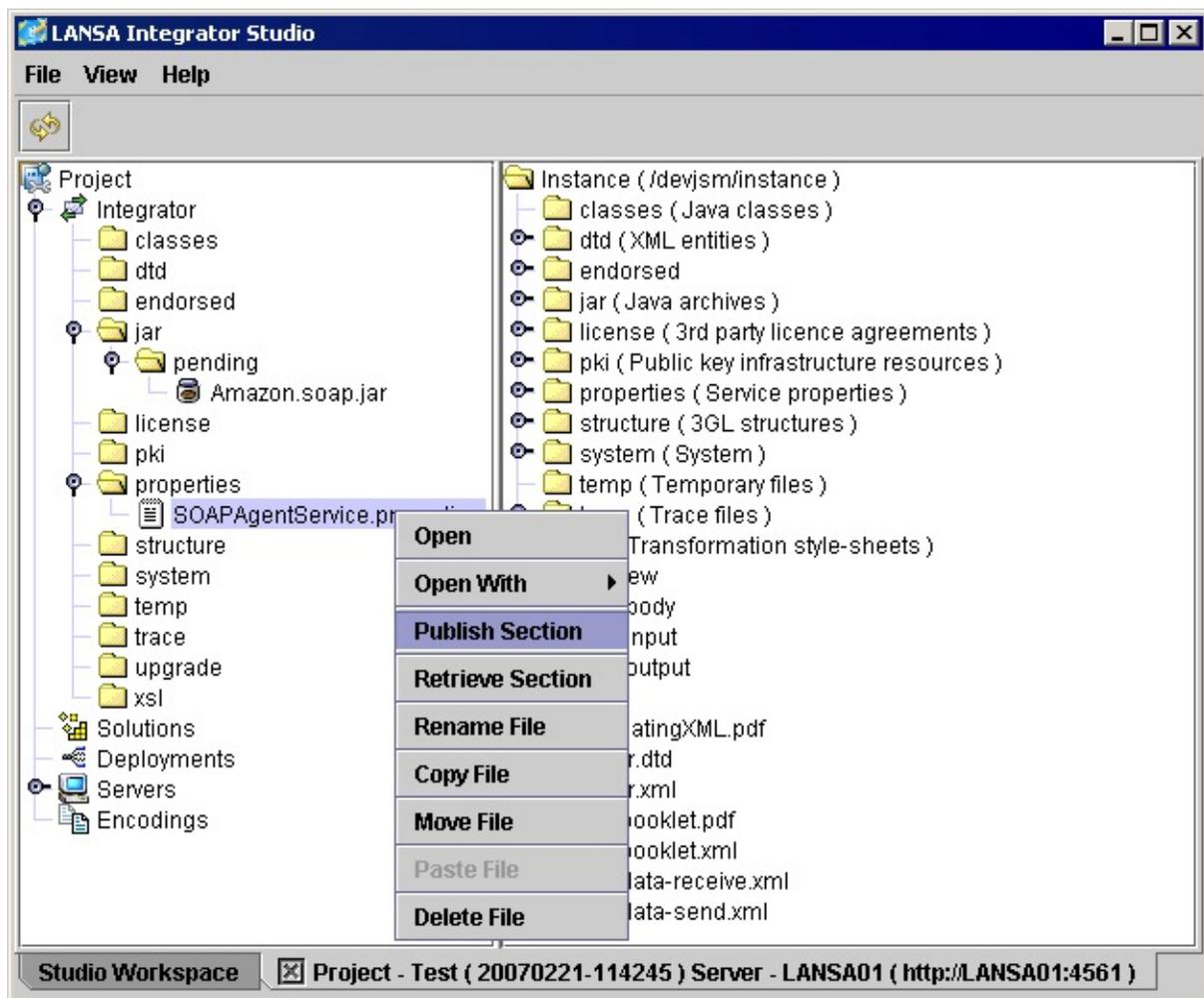
A new panel containing the project is created and made the selected tab.



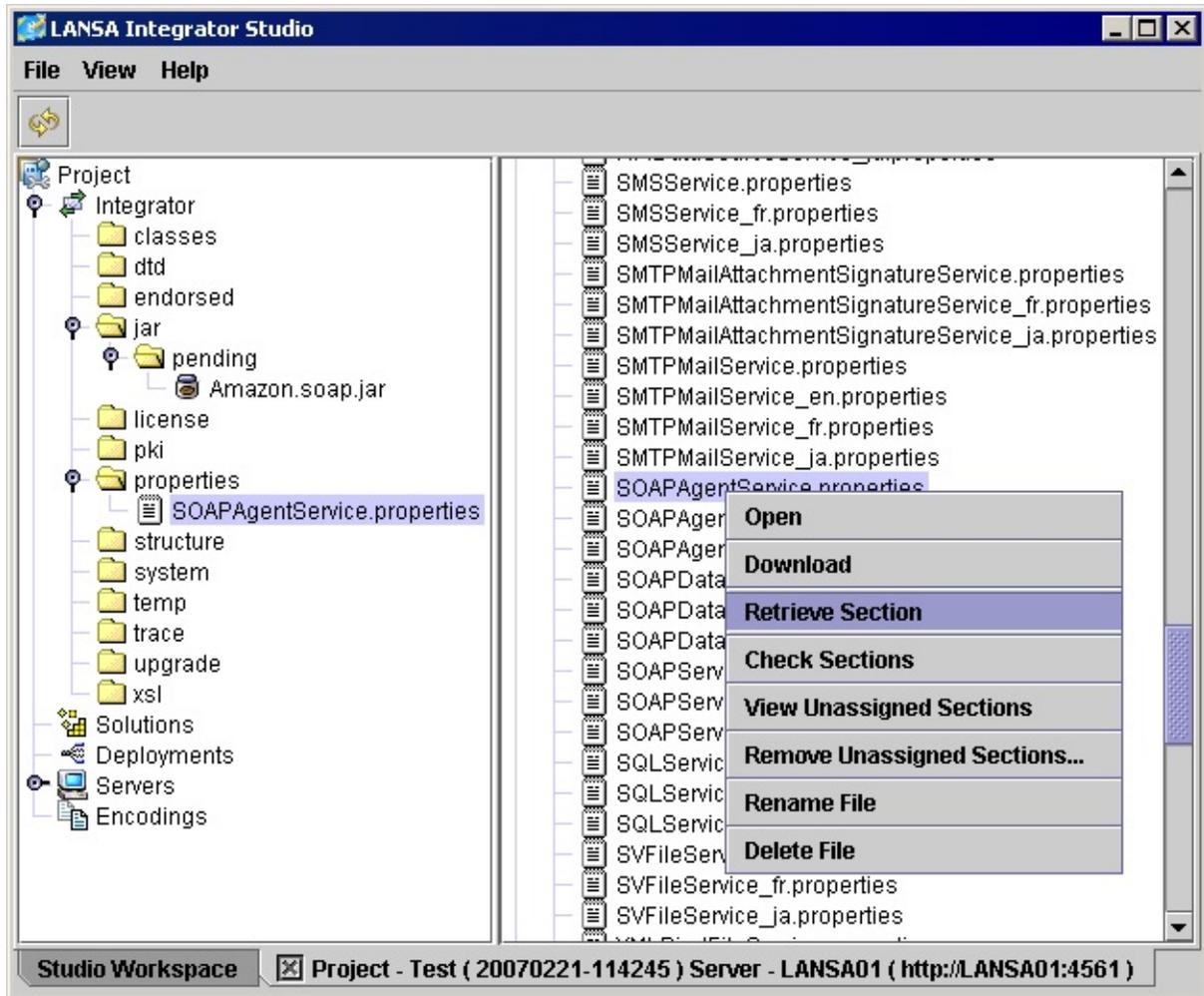
7.6.3 Publish and Retrieve

When a project and server are opened it possible to publish and retrieve sections from the server's service properties files. If the file is not a properties file or a properties file not located in the properties or system sub directory then the whole file is copied across to the server instance directory. The file's directory structure is maintained and child sub directories are created.

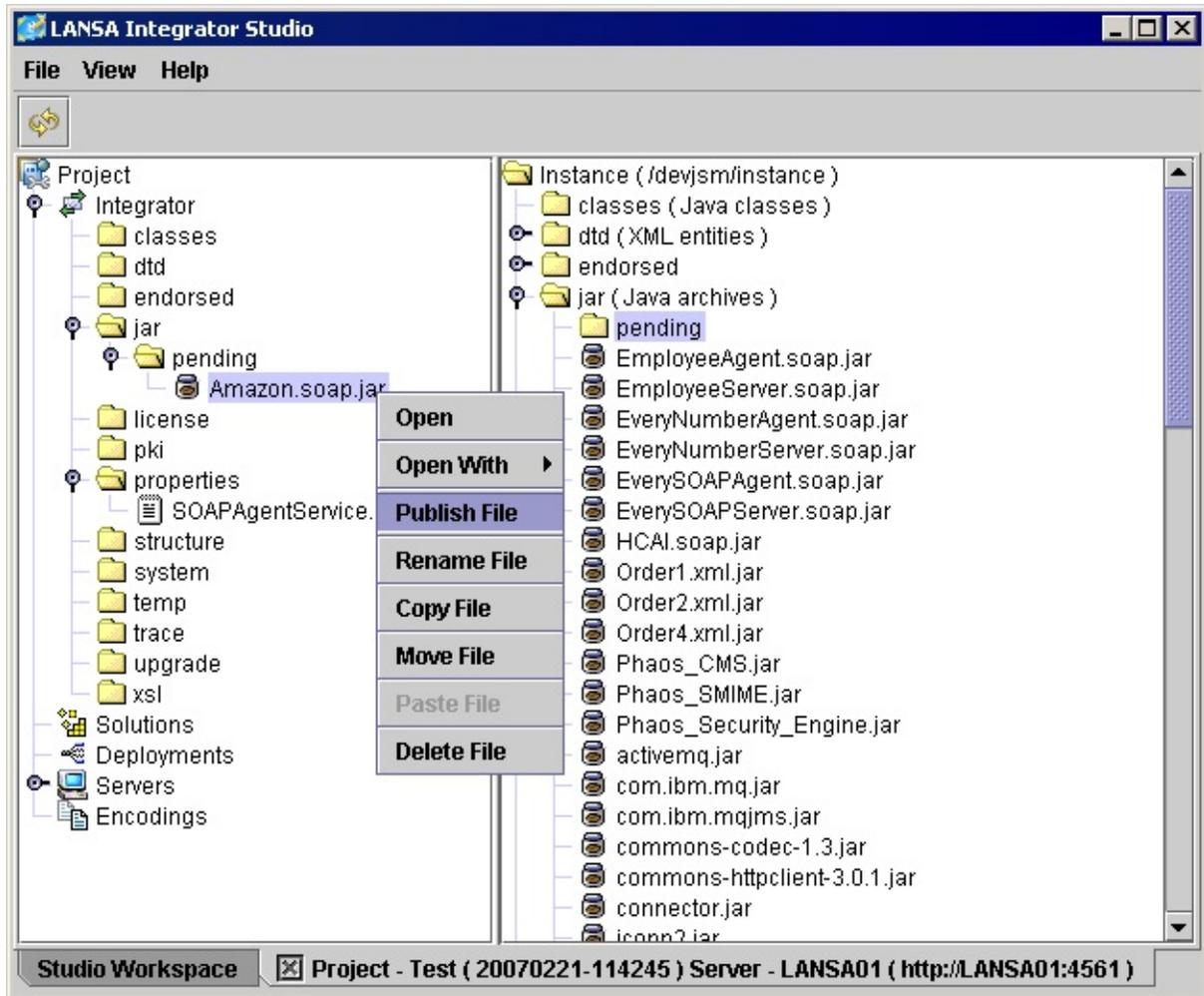
The following image illustrates how the SOAPAgentService.properties entries are published to the server's SOAPAgentService.properties file.



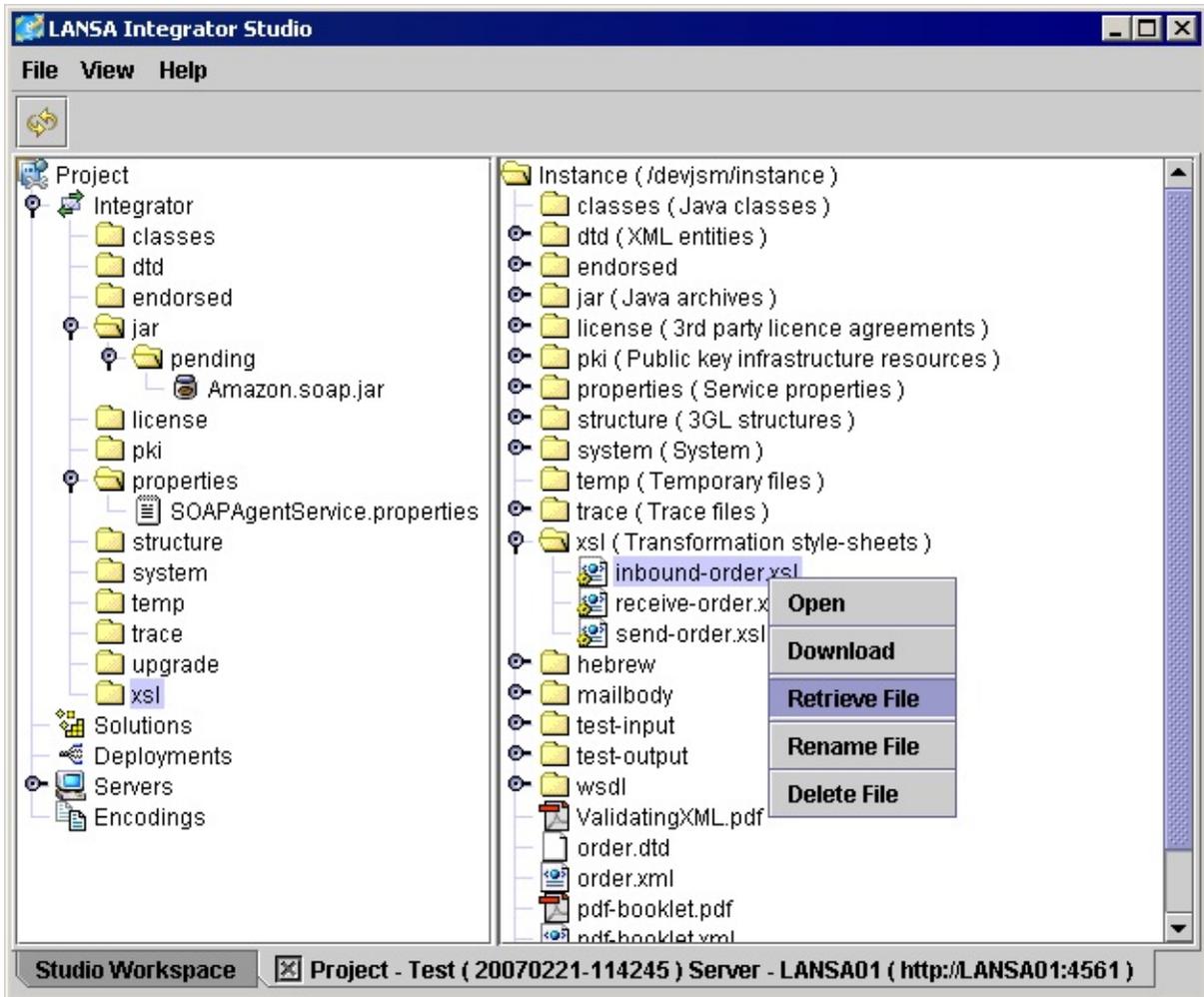
It is also possible to retrieve a project's properties section by selecting the properties file from the server panel.



The following image illustrates how to publish a file to the server.

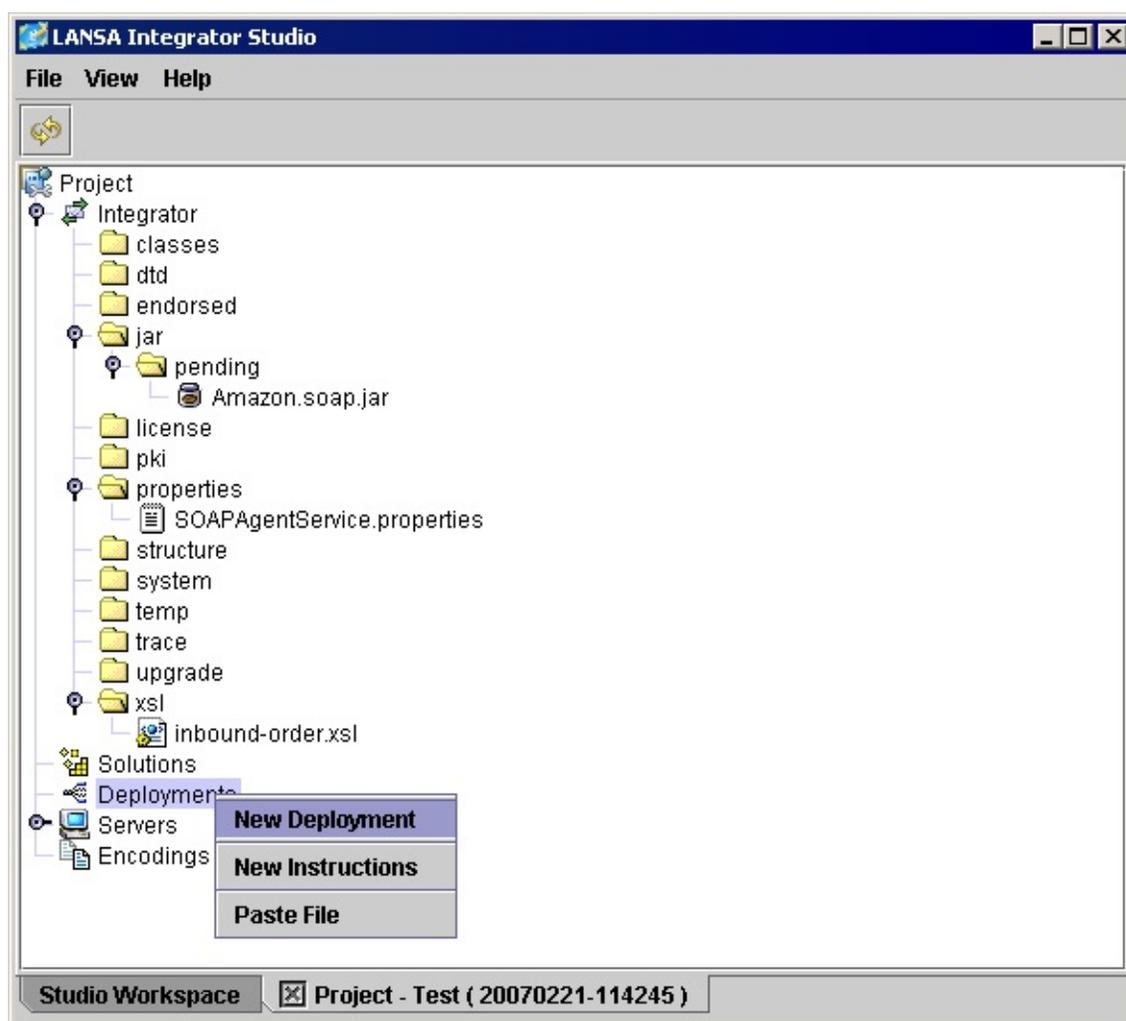


The following image illustrates how to retrieve a file from the server.



7.6.4 Deployment

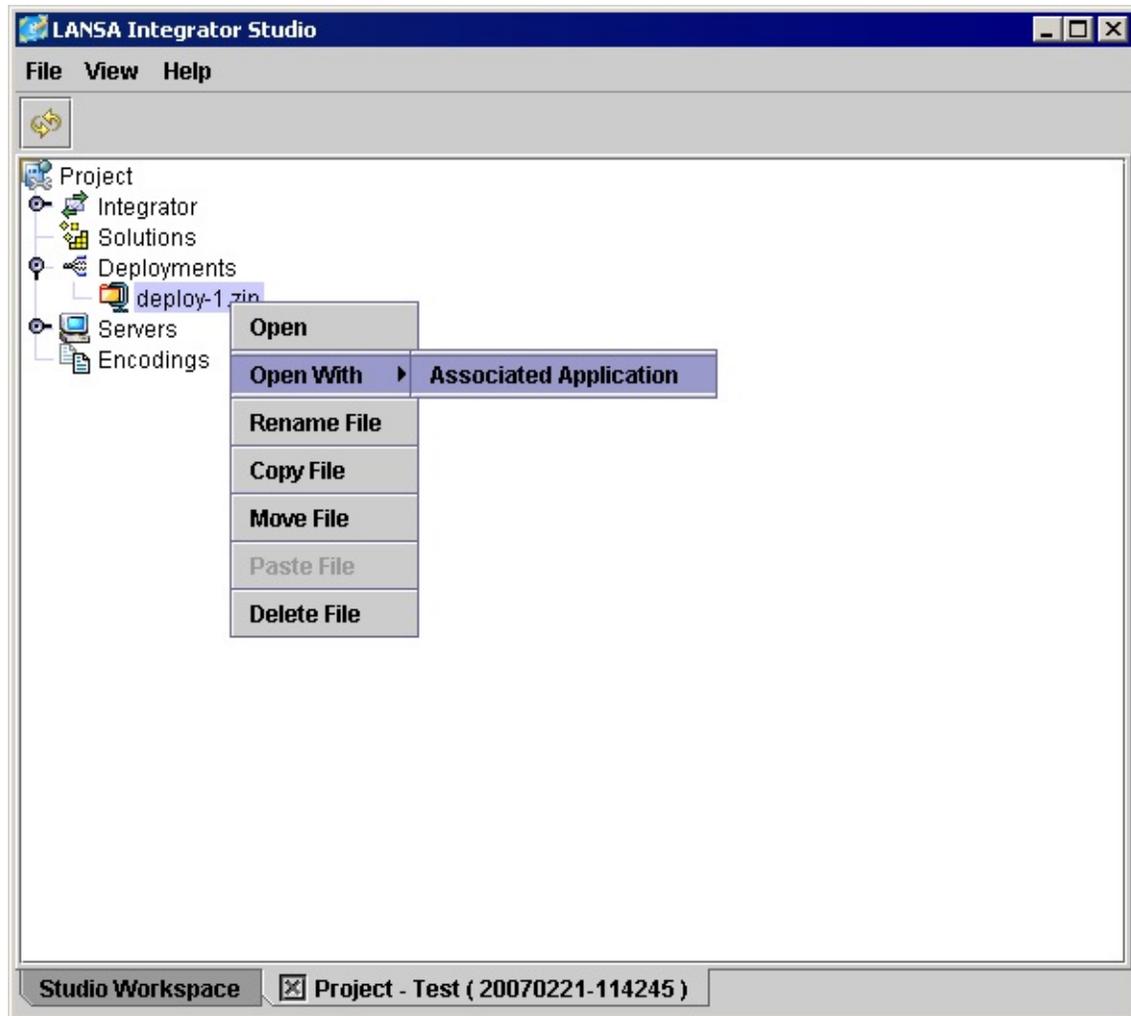
It is good practice to keep project files that need to be published to a server in the project's integrator folder for individual publishing during the development stage. When deploying all these files to a server for testing or production it is good practice to create a deployment file.



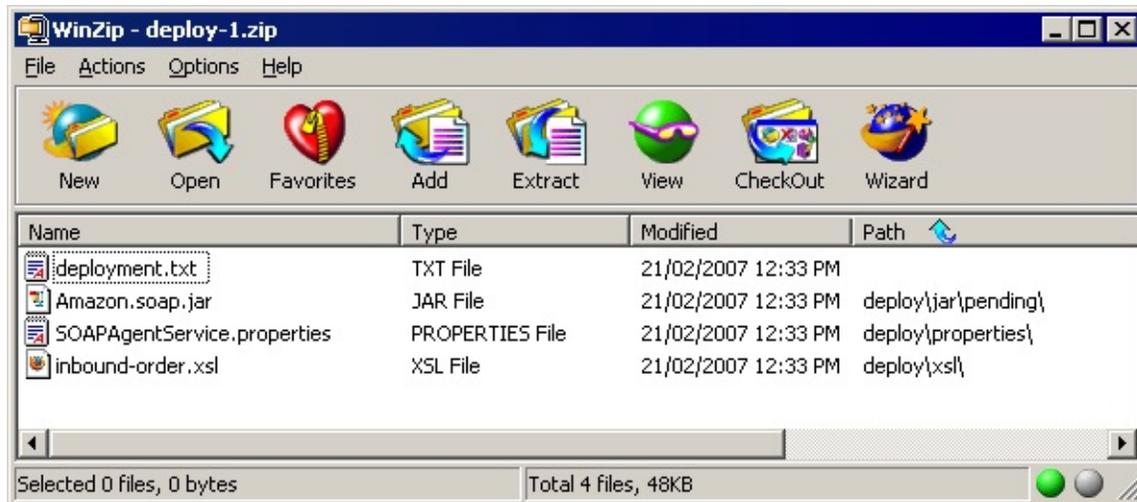
Enter the name of the new deployment file. The '.zip' extension will be added automatically.



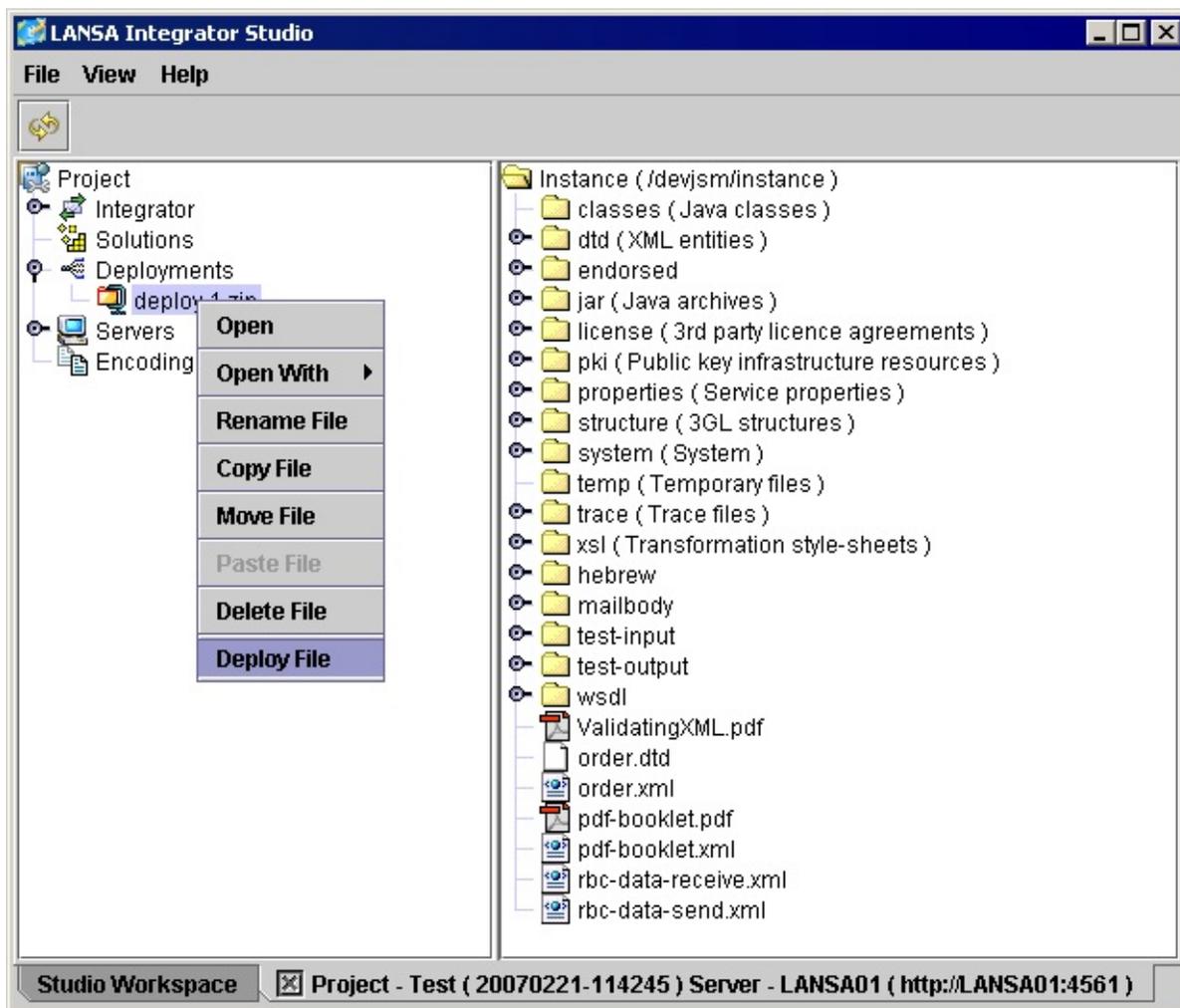
The deployment archive file contains all files from the integrator folder.



An additional 'deployment.txt' file is added as an audit.



To publish the deployment file, open a server instance and select the deploy file menu options.



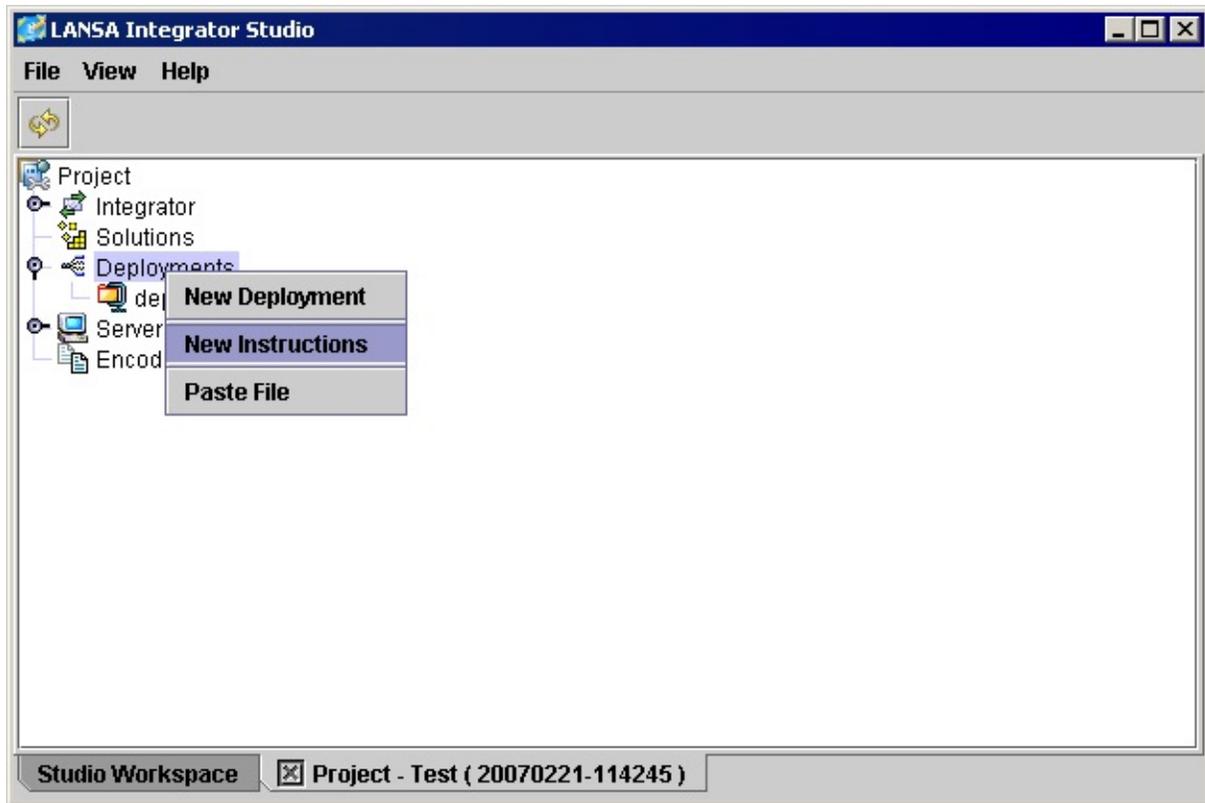
Part of a deployment could require files and directories to be deleted from the

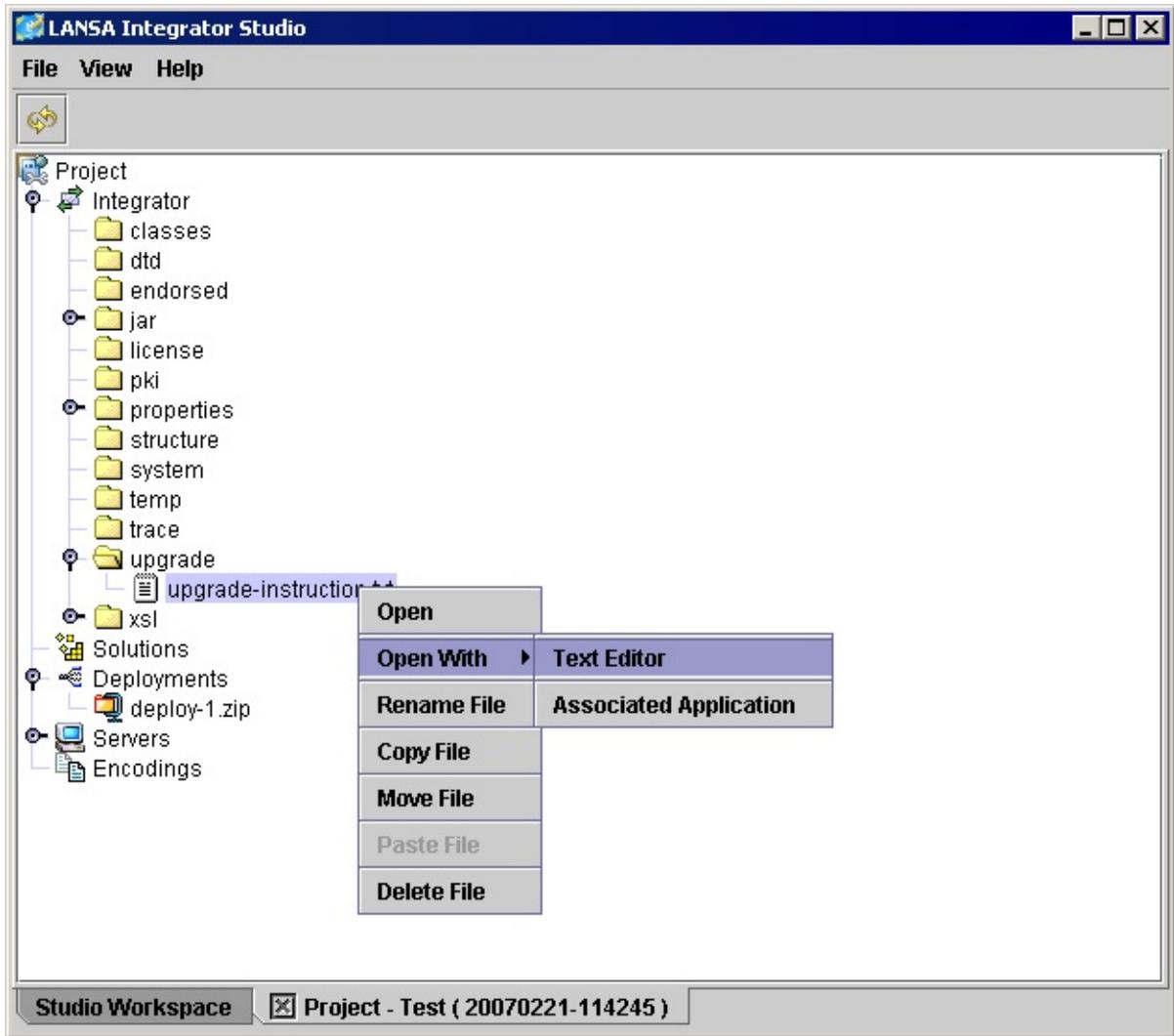
server instance.

An upgrade-instruction.txt file can be created to add instructions to delete files and directories.

The upgrade instructions need to be created before creating the deployment file so this file is included in the deployment.

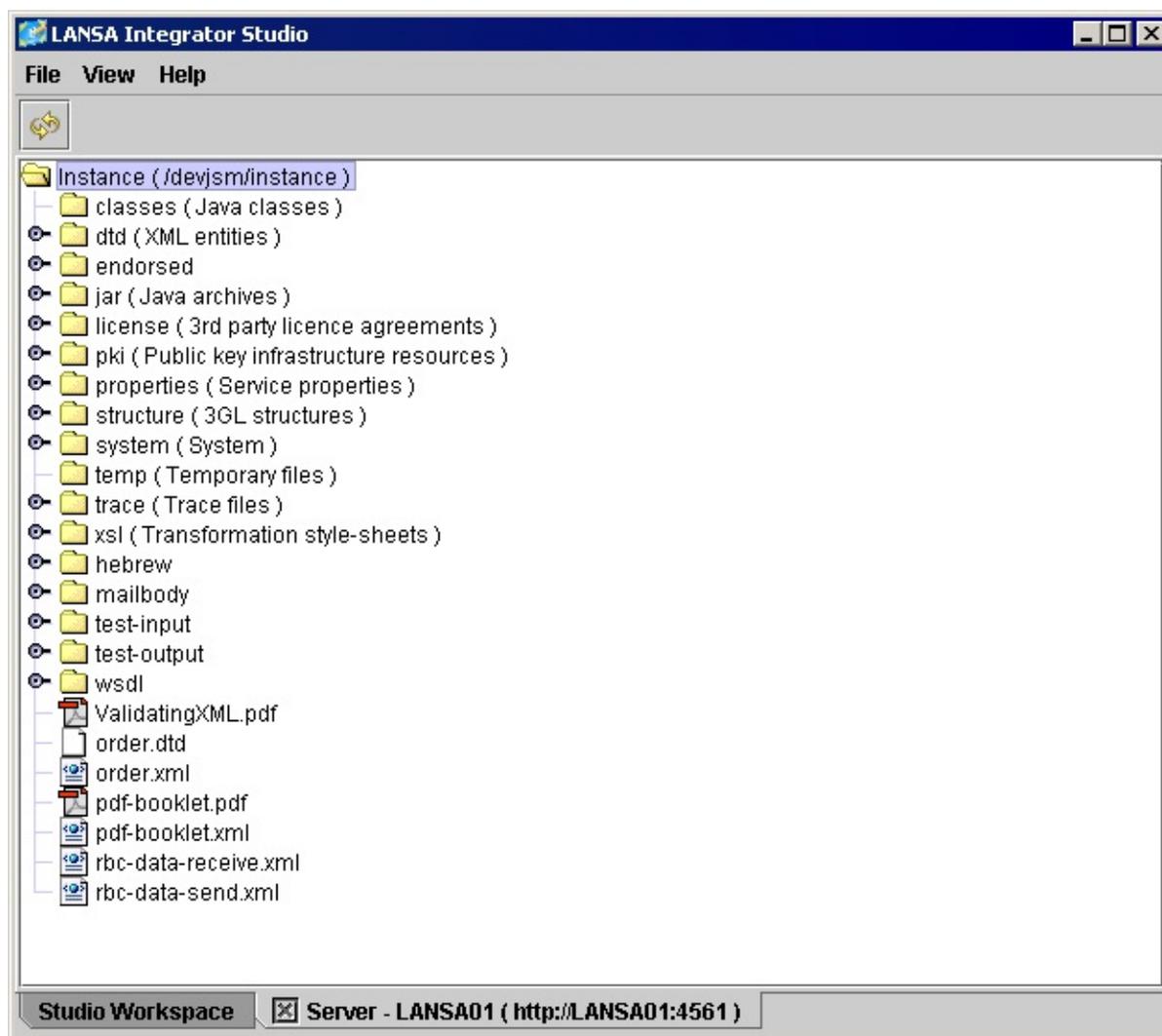
After the deployment file as been deployed, the JSM instance needs to be rebooted so the upgrade instructions are applied to the instance.





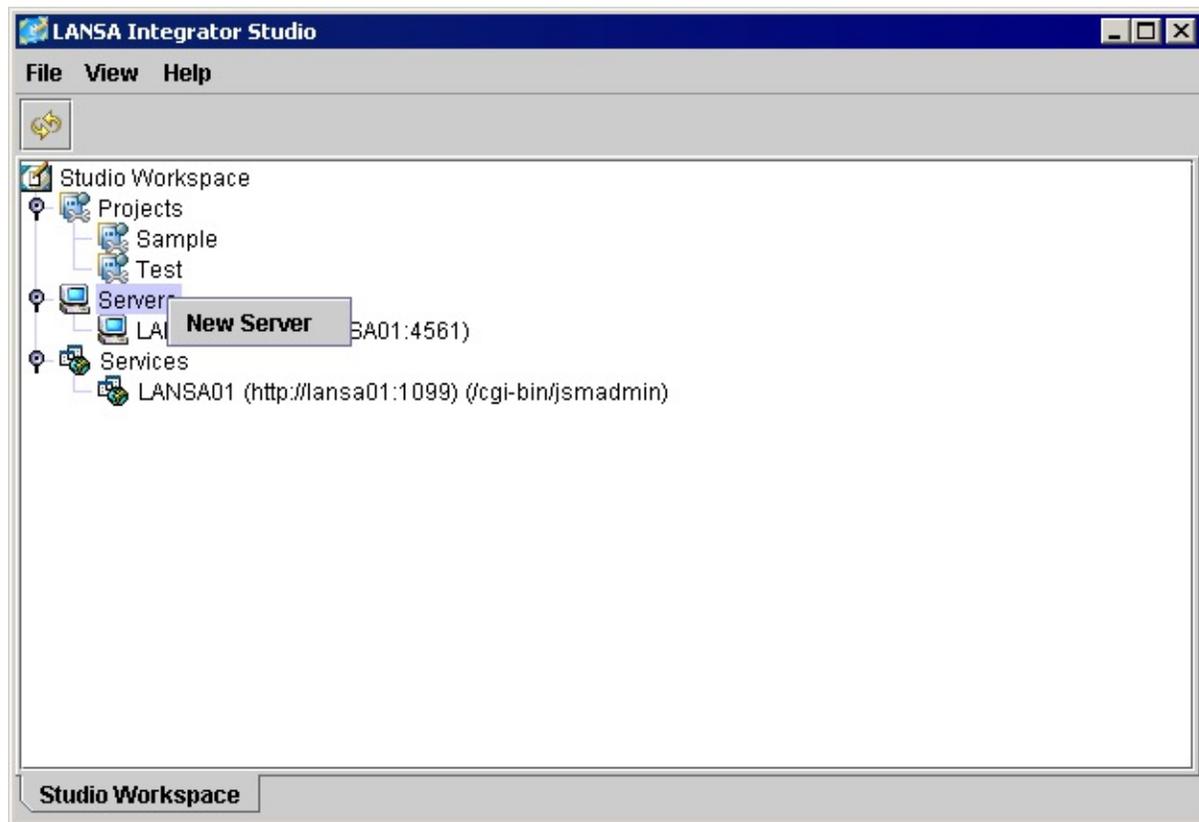
7.7 Servers

Studio allows you to view the contents of the instance directory of the JSM server and to transfer files to and from the local machine. Usually an instance is associated with a Studio project, although you can also open an instance directly without opening a project.



7.7.1 New Server

To create a new server, right click the Servers node in the Studio Workspace panel.



Enter the IP address or domain name of the JSM server. The default port is 4561.



Enter the server name that will appear in the Studio Workspace Servers section.



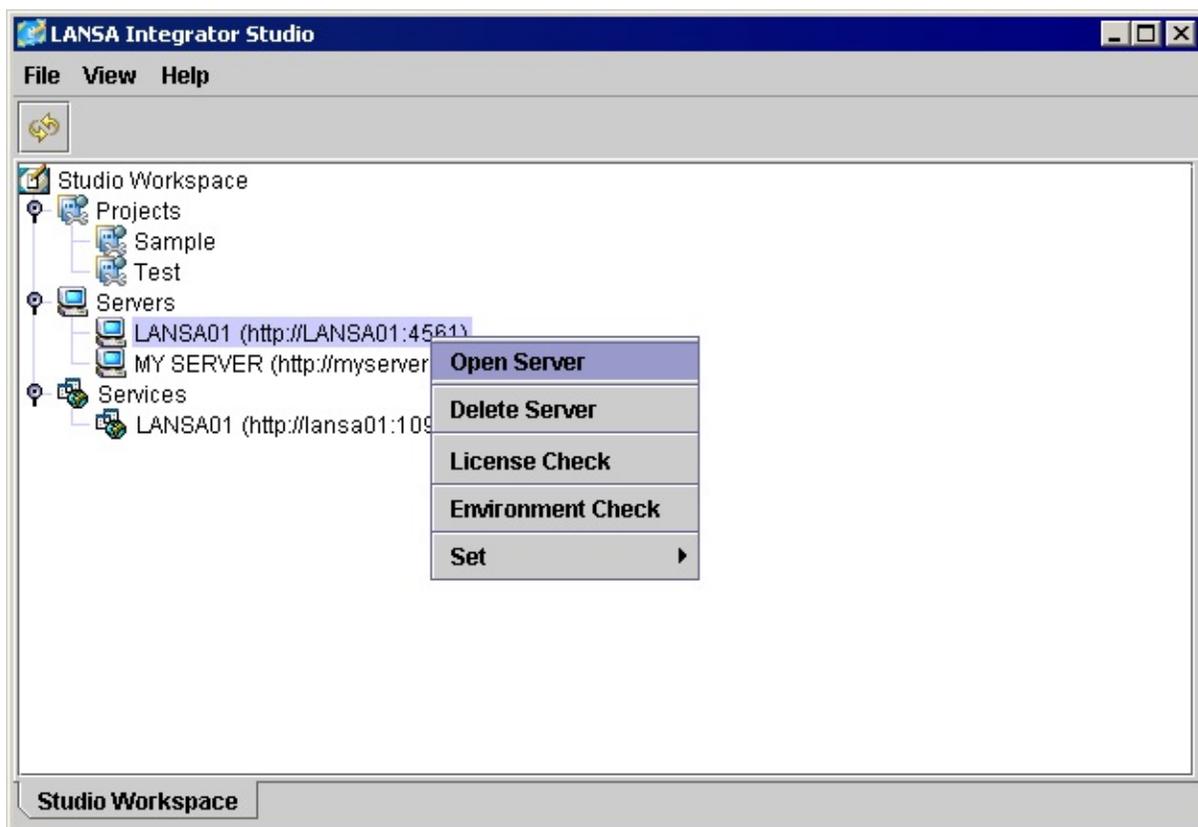
The JSM server does not need to be running to create a new server entry.

7.7.2 Open Server

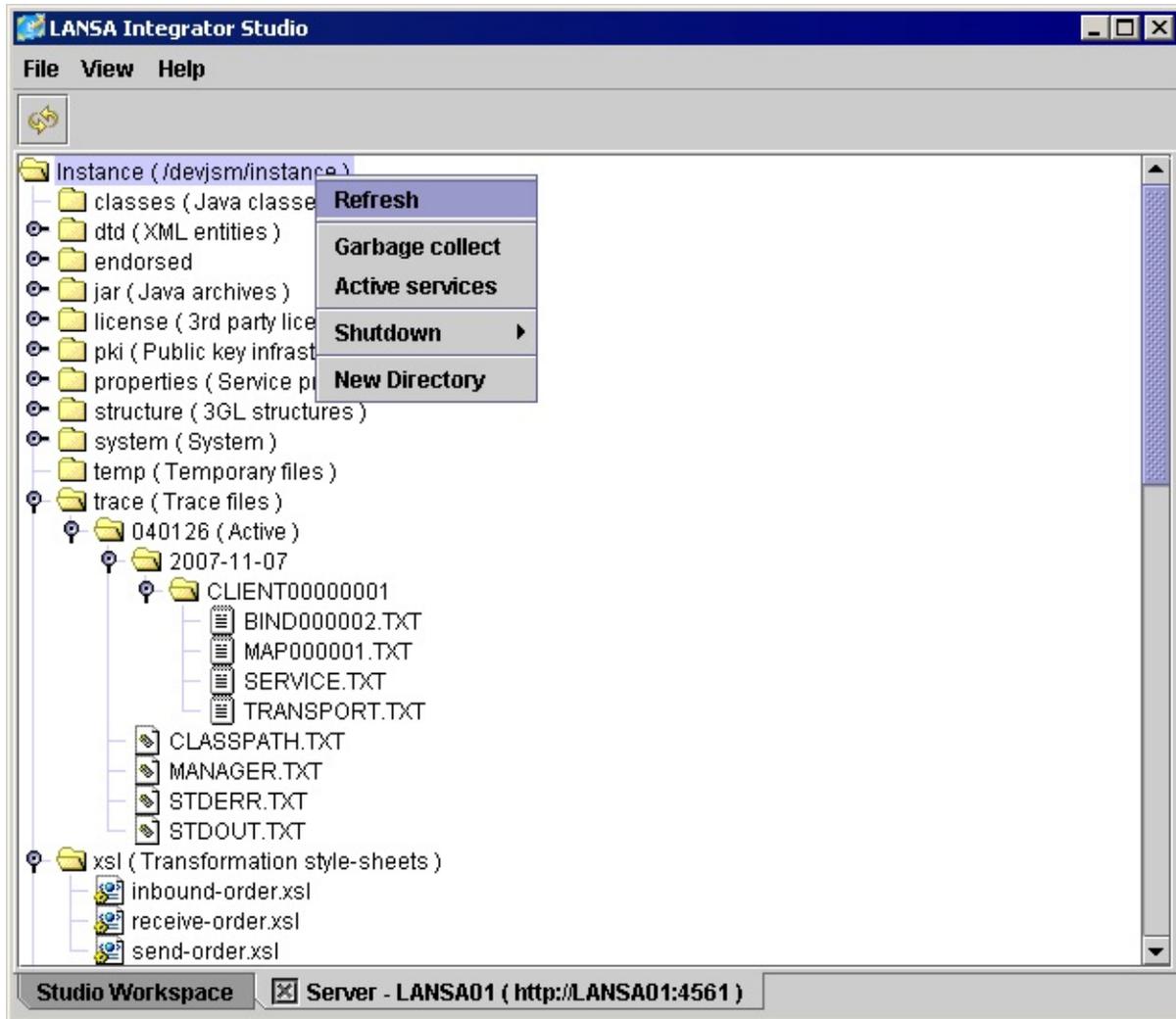
From the Studio Workspace panel select the server to open and double-click or use the pop-up menu to open the server.

The JSM server needs to be running for the server to be opened.

The instance's `manager.properties` file should also contain entries for `studio.client.address` and possibly `studio.authentication` (see [Java Service Manager Console](#)). Also, check that the installed versions of Studio and the JSM are the same (in particular, check that the `jsmide.jar` file is the same).



A new panel containing the opened server is created and made the selected tab.



To copy one of these files to the local file system, either drag the file from Studio and drop it on the Windows desktop or an Explorer window.

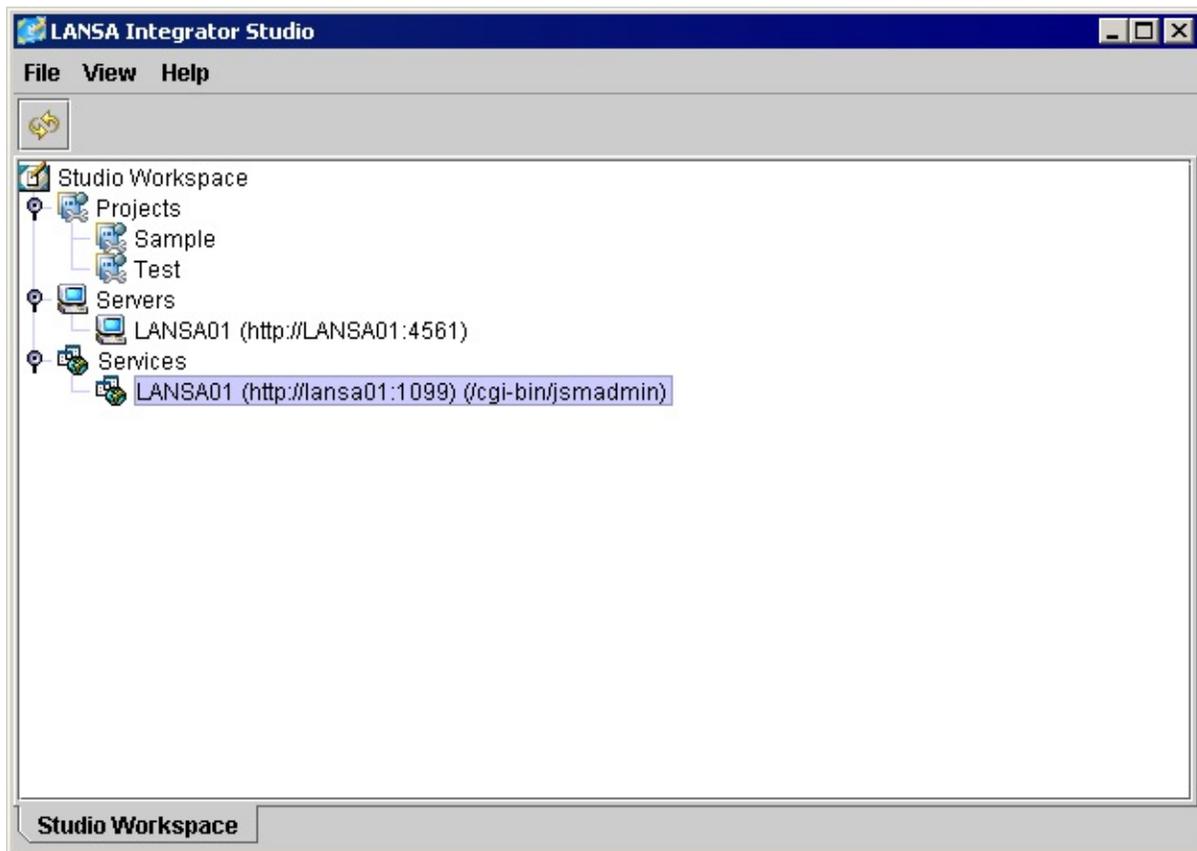
7.8 Services

The HTTP programs JSMDirect and JSMProxy use database files to associate service application to LANSAs function. Studio allows you to maintain these database files.

On IBM i, the DC@W29 and DC@W30 files are used.

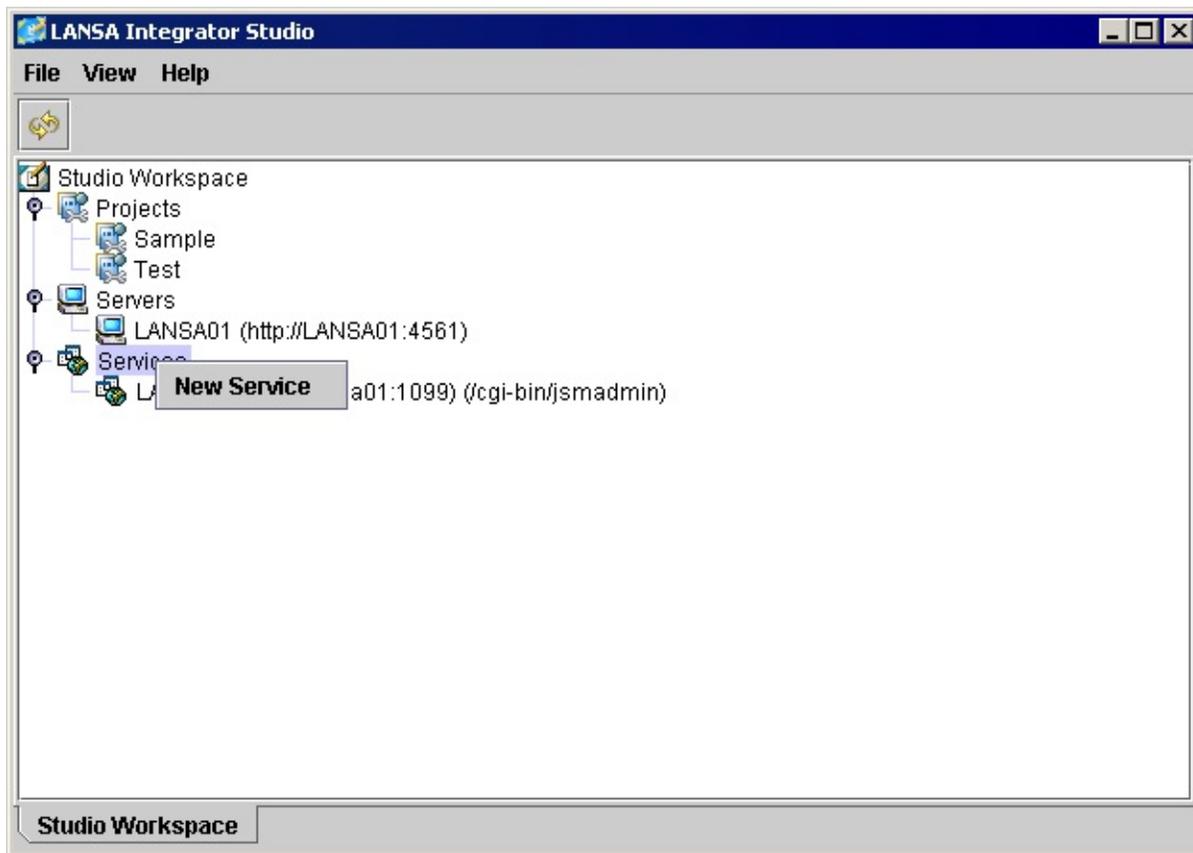
On Linux and Windows, files dc_w29.txt and dc_w30.txt are used.

Studio accesses these files using the HTTP server program JSMAdmin.



7.8.1 New Service

To create a new service, right click the Services node in the Studio Workspace panel.



Enter the IP address or domain name of the HTTP server. The default port is 80.



Enter the jsadmin program to be used to access the database files. The default program is /cgi-bin/jsadmin. Use /cgi-bin/jsadmin.exe for Windows IIS.



Enter the service name that will appear in the Studio Workspace Services section.



The HTTP server does not need to be running to create a new service.

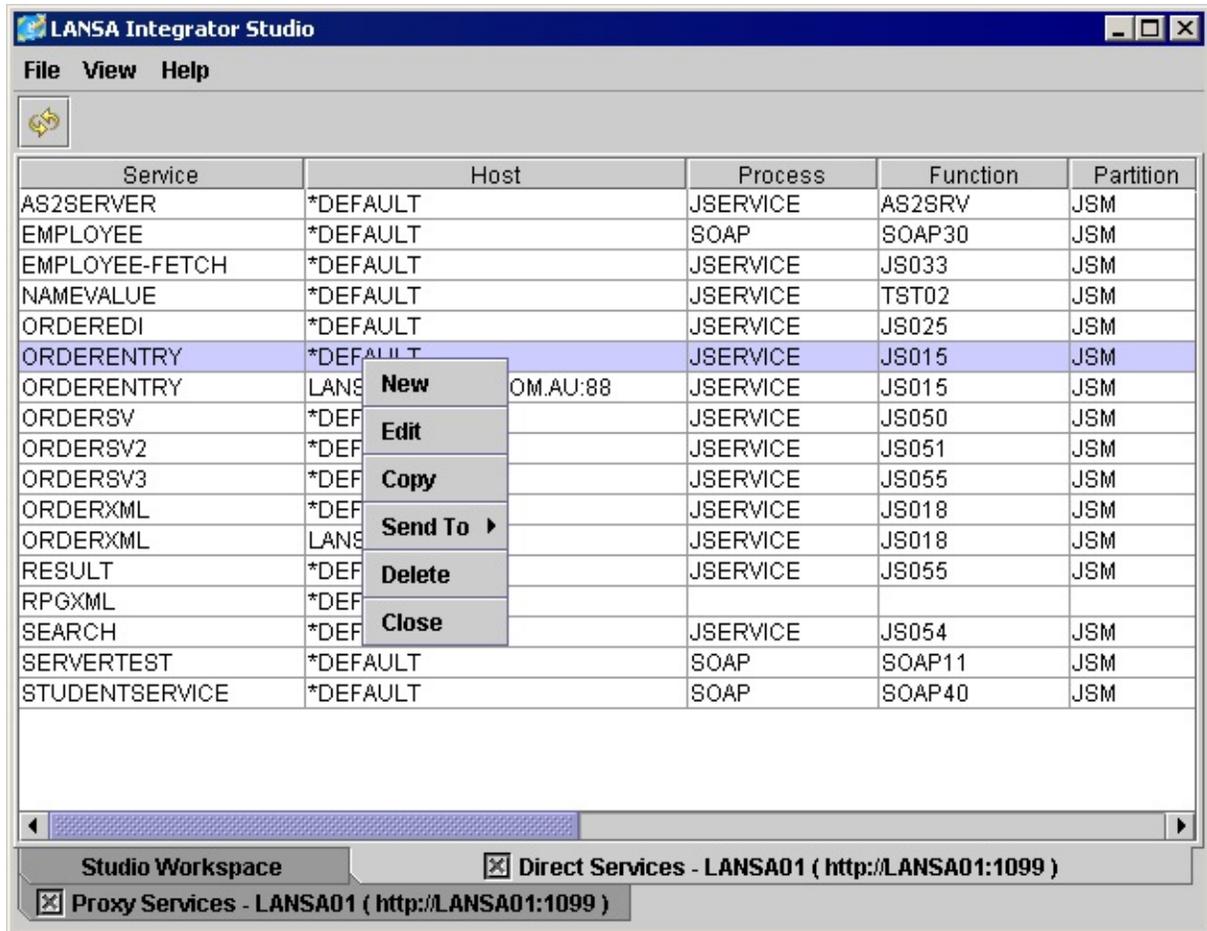
7.8.2 Open Service

From the Studio Workspace panel select the service to open and double-click or use the pop-up menu to open the service.

The HTTP server needs to be configured and running for the server to be opened.

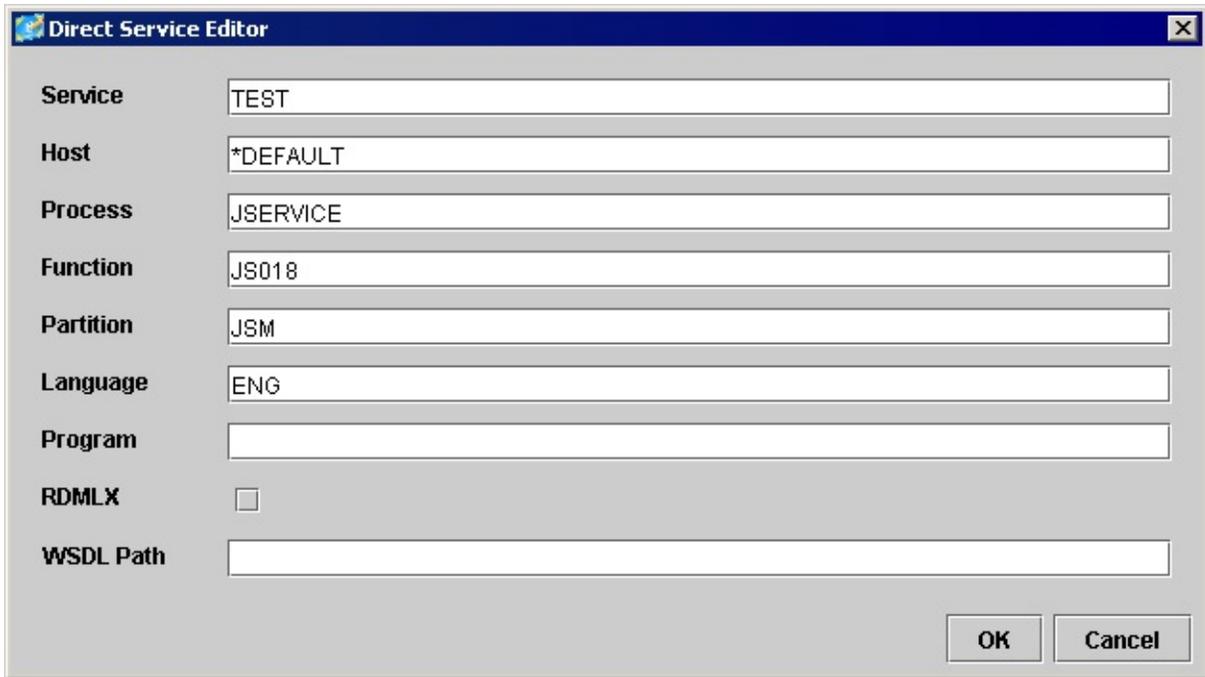


New panels containing the JSMDirect services and JSMProxy services are created and added to the tabbed pane.



New

The New option will allow you to create a new database entry.



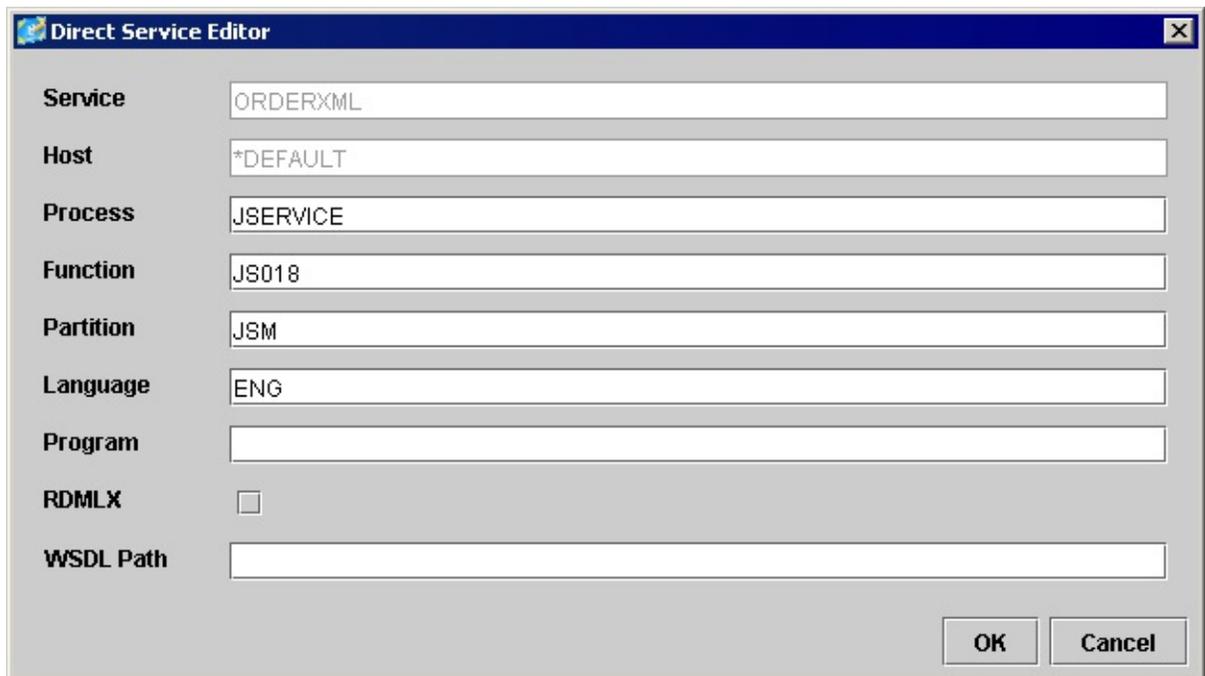
The screenshot shows a dialog box titled "Direct Service Editor" with a close button (X) in the top right corner. The dialog contains several input fields and a checkbox:

Service	TEST
Host	*DEFAULT
Process	JSERVICE
Function	JS018
Partition	JSM
Language	ENG
Program	
RDMLX	<input type="checkbox"/>
WSDL Path	

At the bottom right, there are two buttons: "OK" and "Cancel".

Edit

The *Edit* option will allow you to modify the currently selected entry. The Service and Host may not be changed. The values entered are not validated.



The screenshot shows the same "Direct Service Editor" dialog box, but with the "Service" field containing "ORDERXML". All other fields and the "RDMLX" checkbox remain the same as in the previous screenshot.

Service	ORDERXML
Host	*DEFAULT
Process	JSERVICE
Function	JS018
Partition	JSM
Language	ENG
Program	
RDMLX	<input type="checkbox"/>
WSDL Path	

At the bottom right, there are two buttons: "OK" and "Cancel".

Copy

The *Copy* option will copy the currently selected entries to the clipboard, in

comma-separated format. This can be pasted in any text editor, or in an email.

Send To

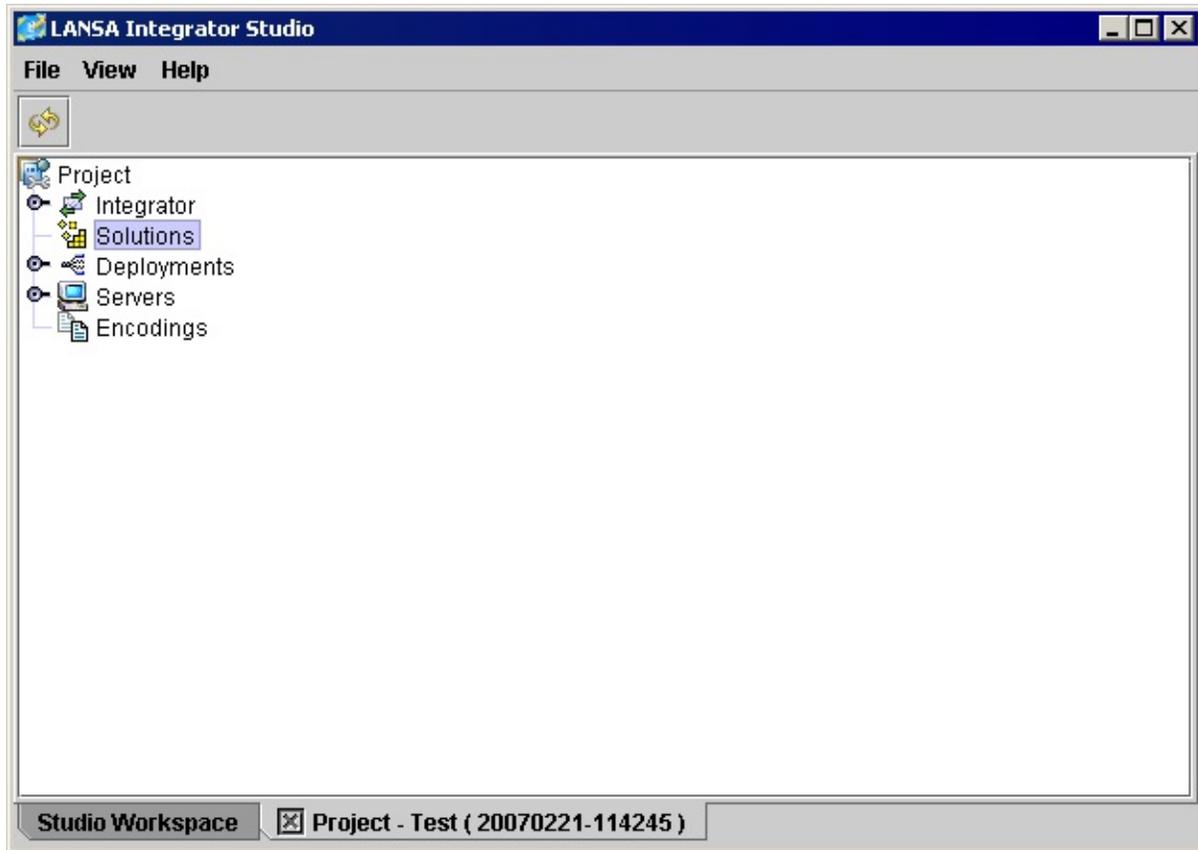
The *Send To* option will copy the selected entry to another database.

Delete

The *Delete* option will remove the currently selected entry. You will be asked to confirm the deleted record.

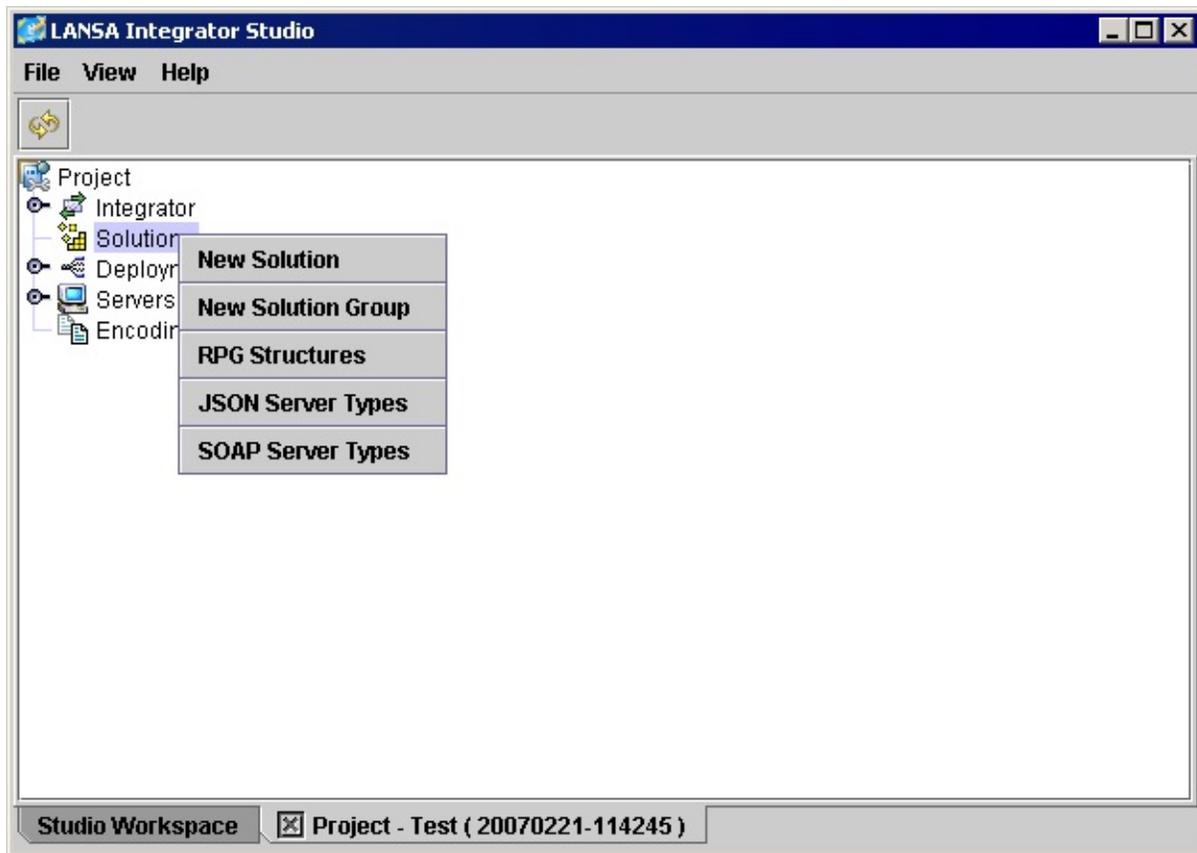
7.9 Solutions

Within a Studio project one or more solution group folders can be created. These group folders contain the files used by the Studio tools.

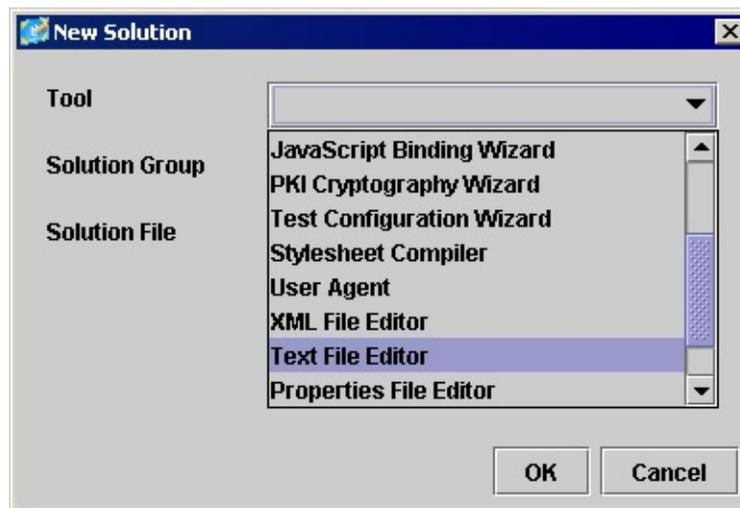


7.9.1 New Solution

To create a new solution, right click the Solution node in the Project panel.



From the Tool combo, selected the required tool, in this example the Text File Editor has been selected.



Because the Solution folder contains no group folders, the Group Combo is

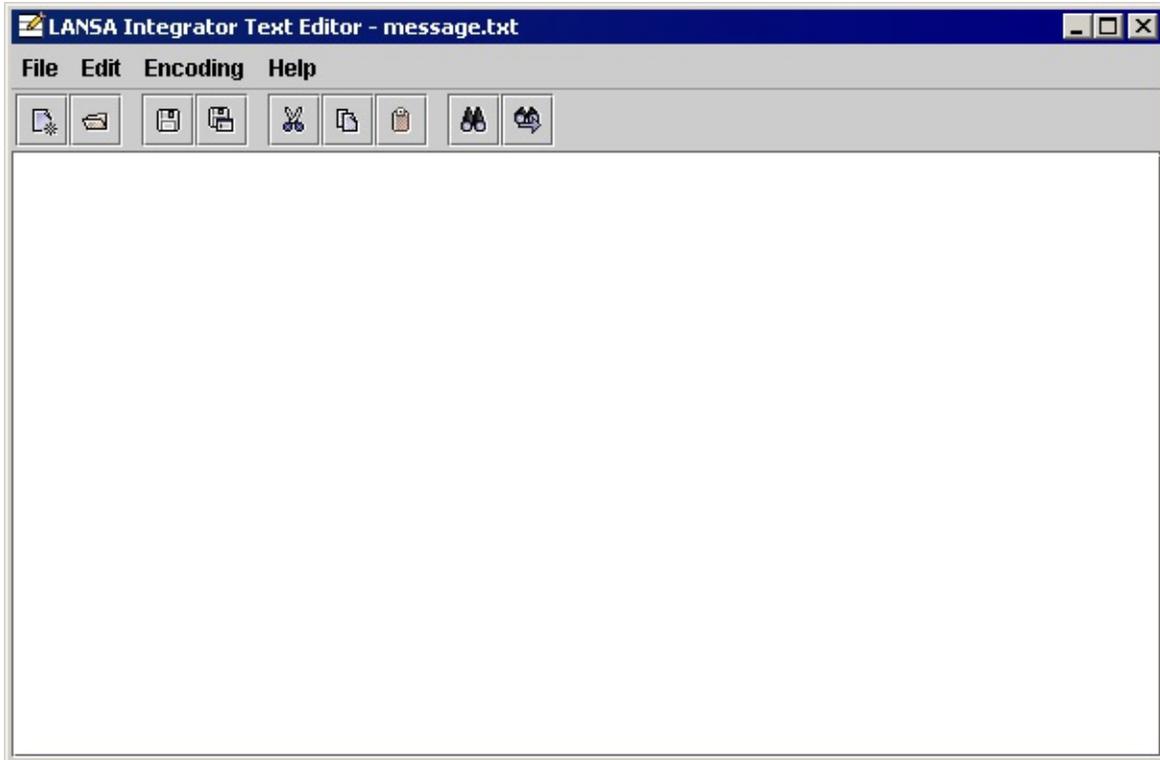
empty, enter the name of the group folder that will be created.
Enter the name of the file to be used by the Text File Editor there is no need to add an extension.



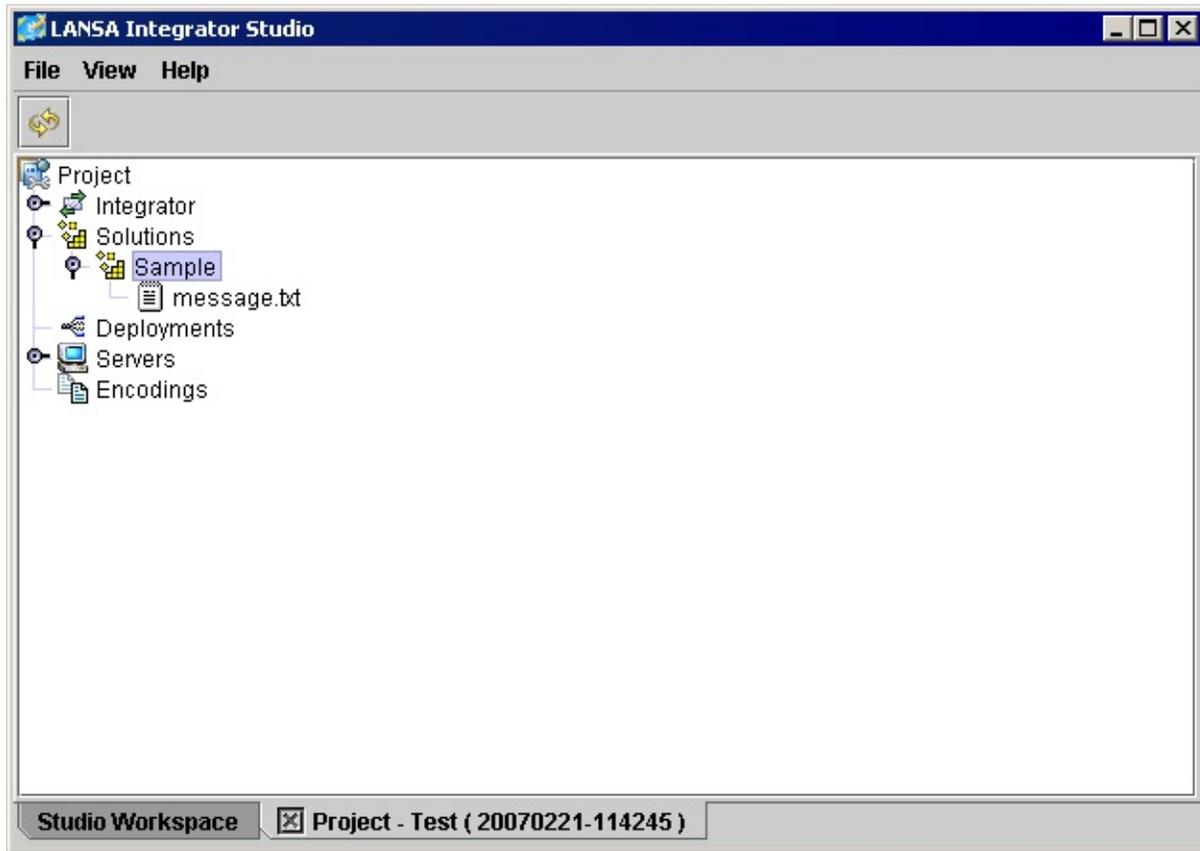
Because the Group folder does not exist you will be prompted to confirm its creation.



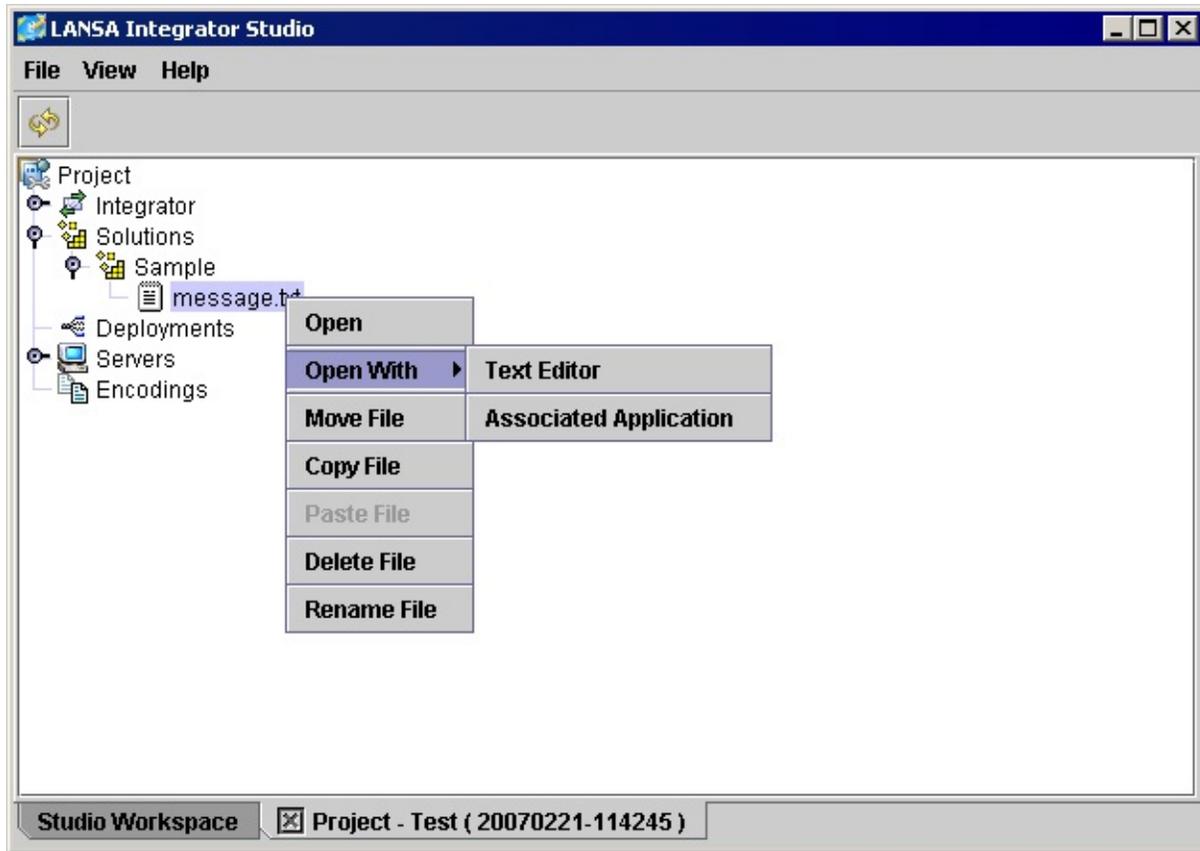
The Sample folder is created and the message.txt is also created and opened by the Text File Editor.



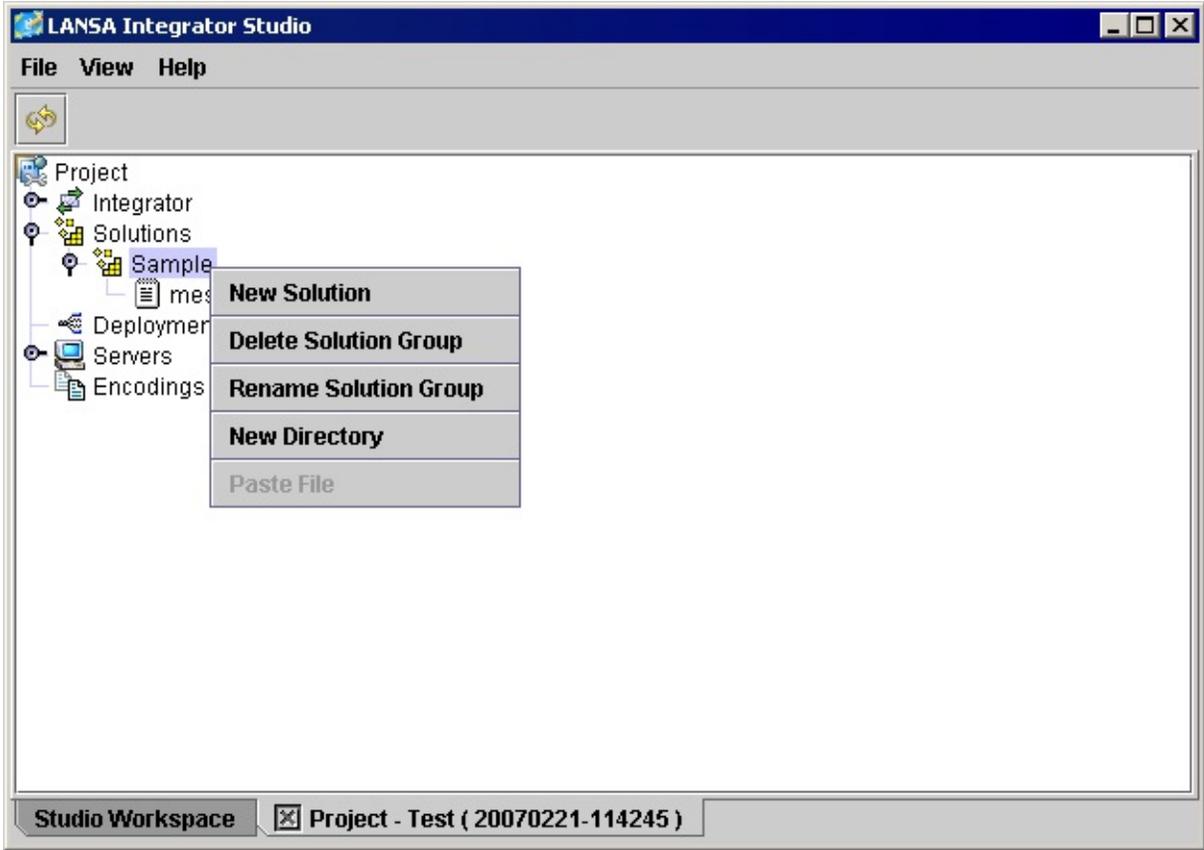
When the group folder is created, several sub folders are also created. These folders are used to manage content created by SOAP Wizard, XML Binding Wizard, JSON Binding Wizard and XSL Compiler tools.



Once a file is created or copied into the group folder, Studio can be used to maintain that resource.



A solution can also be created from the group folder node. In this case the Group combo will be set to the selected Group folder.



7.10 XML Editor

The XML Editor is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The XML Editor allows files to be converted from one encoding to another encoding.

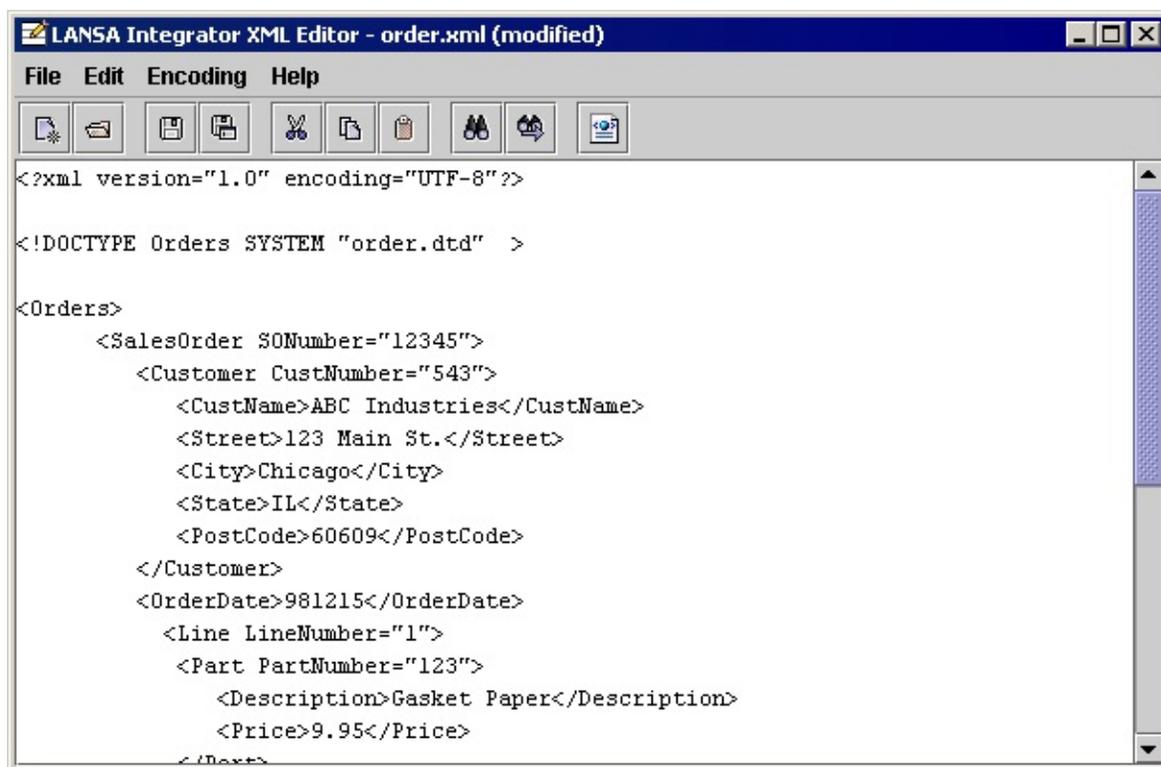
When reading a file, the XML editor can auto-detect the encoding of the XML source and convert the byte stream to Unicode characters.

If an encoding has been selected, this value will be used to convert the byte stream to Unicode characters and the XML declaration will be modified to suit the selected encoding.

When saving to a file and the encoding is auto-detect, the XML declaration encoding will be used to select a Java encoding to convert the Unicode characters to a byte stream.

If an encoding has been selected, this value will be used to convert the Unicode characters to a byte stream and the XML declaration encoding will be modified to suit the selected Java encoding.

Refer to [IANA Encodings](#) for more details.



The screenshot shows a window titled "LANSA Integrator XML Editor - order.xml (modified)". The window has a menu bar with "File", "Edit", "Encoding", and "Help". Below the menu bar is a toolbar with icons for file operations (new, open, save, print, copy, paste, undo, redo, find, help). The main text area contains the following XML code:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE Orders SYSTEM "order.dtd" >

<Orders>
  <SalesOrder SONumber="12345">
    <Customer CustNumber="543">
      <CustName>ABC Industries</CustName>
      <Street>123 Main St.</Street>
      <City>Chicago</City>
      <State>IL</State>
      <PostCode>60609</PostCode>
    </Customer>
    <OrderDate>981215</OrderDate>
    <Line LineNumber="1">
      <Part PartNumber="123">
        <Description>Gasket Paper</Description>
        <Price>9.95</Price>
      </Part>
    </Line>
  </SalesOrder>
</Orders>
```

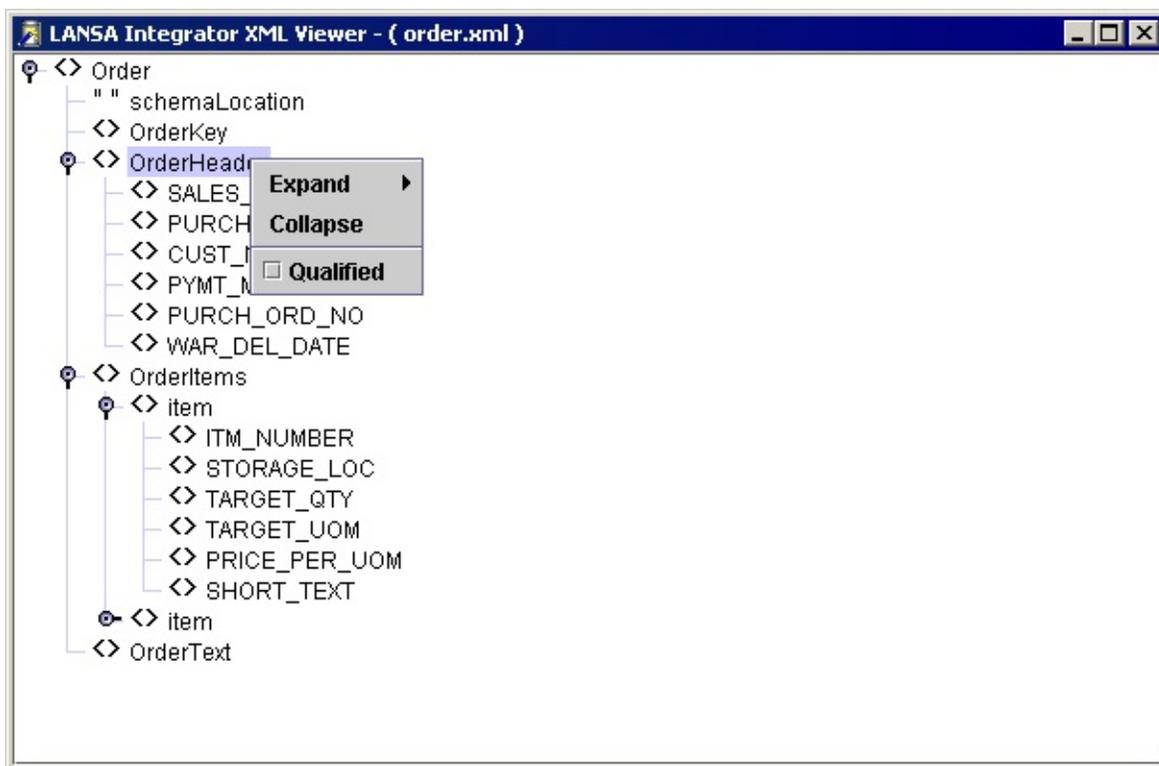
7.11 XML Viewer

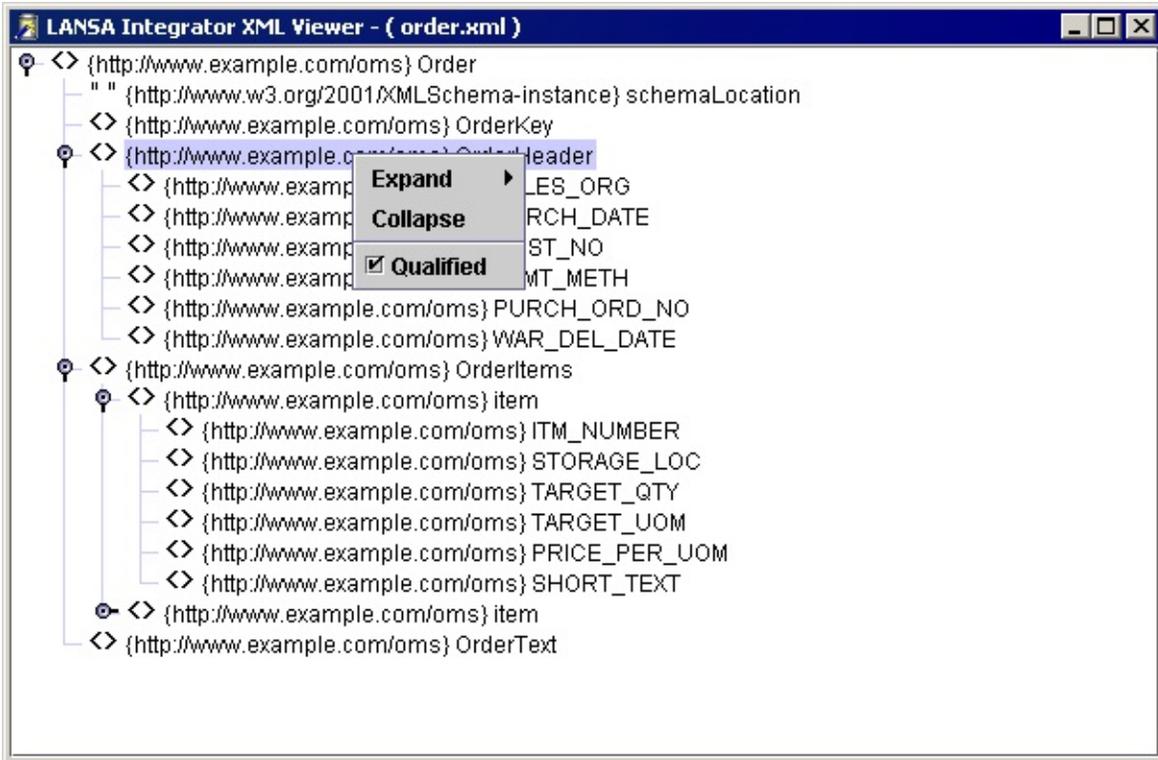
The XML Viewer is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The XML Viewer is used to view XML files in a hierarchical tree format.

The XML Viewer is associated to the file extension '.xml' and is available as an associated application from Studio.

A popup menu is available to expand and collapse elements and to enable the displaying of the associated namespace URI with the element and attribute names.





7.12 XMLSchema Viewer

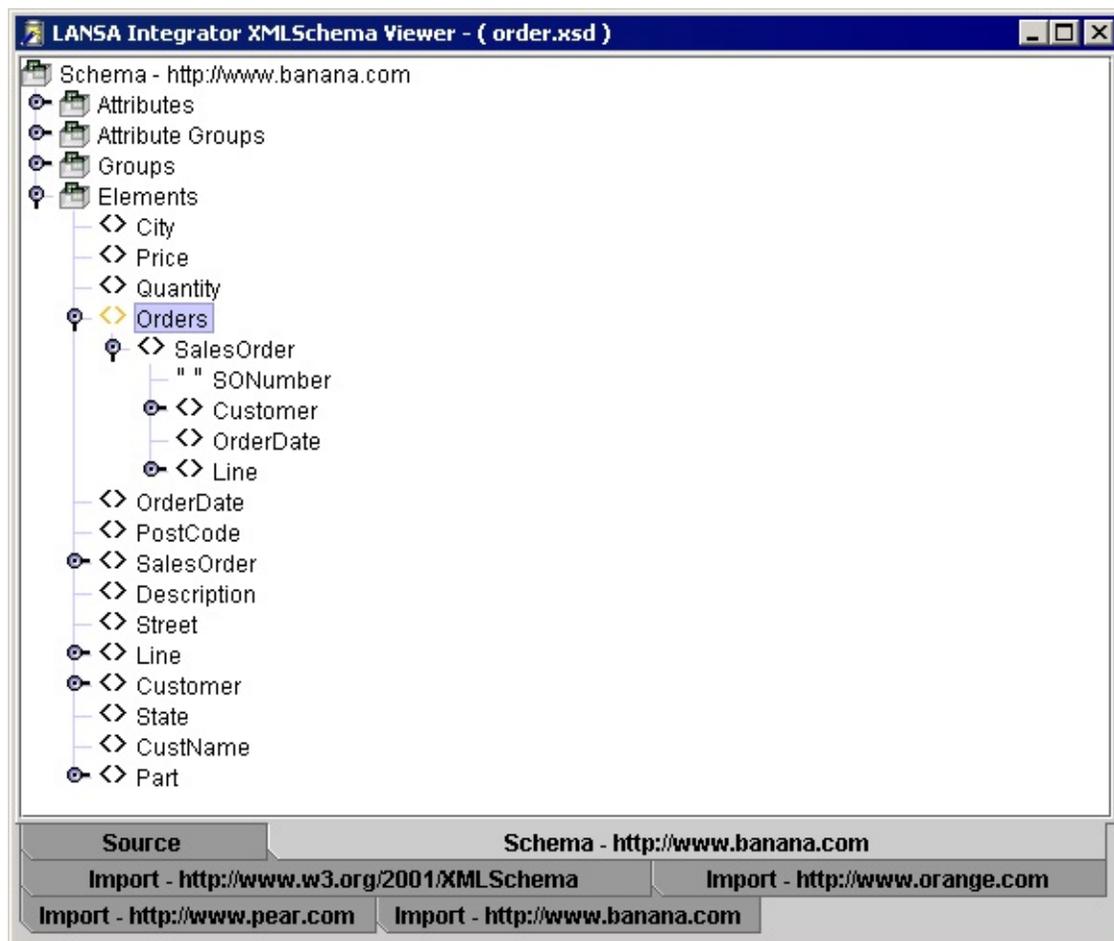
The XMLSchema Viewer is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The XMLSchema Viewer is used to view XML schema files in a hierarchical tree format.

The XMLSchema Viewer is associated to the file extension '.xsd' and is available as an associated application from Studio.

Top-level elements are colored orange.

A popup menu is available to expand and collapse elements, enable the displaying of the associated namespace URI with the element and attribute names and creating a sample of XML.

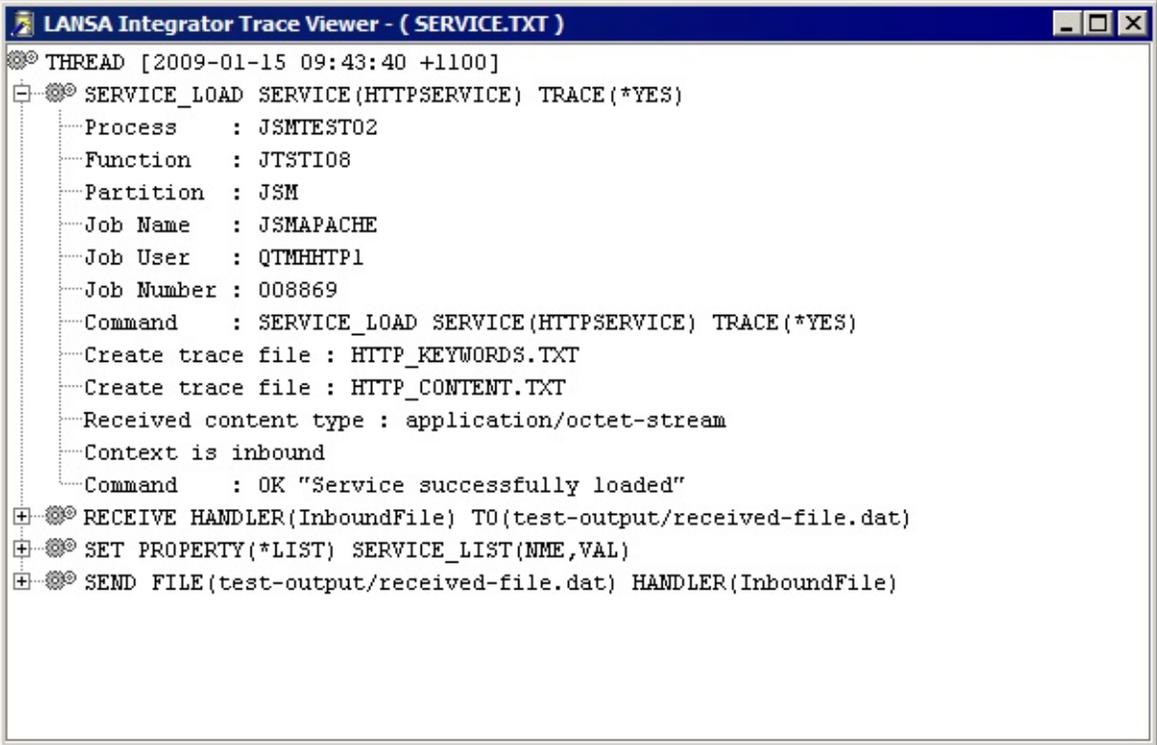


7.13 Trace Viewer

The Trace Viewer is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The Trace Viewer is used to view SERVICE.TXT and TRANSPORT.TXT trace files in a hierarchical tree format.

A popup menu is available to expand and collapse elements.



```
LANSA Integrator Trace Viewer - ( SERVICE.TXT )
THREAD [2009-01-15 09:43:40 +1100]
  SERVICE_LOAD SERVICE(HTTPSERVICE) TRACE(*YES)
    Process      : JSMTEST02
    Function     : JTSTI08
    Partition    : JSM
    Job Name     : JSMPACHE
    Job User     : QTMHHTP1
    Job Number   : 008869
    Command      : SERVICE_LOAD SERVICE(HTTPSERVICE) TRACE(*YES)
    Create trace file : HTTP_KEYWORDS.TXT
    Create trace file : HTTP_CONTENT.TXT
    Received content type : application/octet-stream
    Context is inbound
    Command      : OK "Service successfully loaded"
  RECEIVE HANDLER(InboundFile) TO(test-output/received-file.dat)
  SET PROPERTY(*LIST) SERVICE_LIST(NME,VAL)
  SEND FILE(test-output/received-file.dat) HANDLER(InboundFile)
```

```
LANSA Integrator Trace Viewer - ( TRANSPORT.TXT )
+⊗ THREAD [2009-01-15 09:43:40 +1100] [265 ms]
+⊗ SERVICE_OPEN [30 ms]
-⊗ SERVICE_LOAD SERVICE(HTTPSERVICE) TRACE(*YES) [19 ms]
+⊗ Protocol
+⊗ Build Command ( 2 ms )
+⊗ Execute Command ( 17 ms )
+⊗ RECEIVE_HANDLER(InboundFile) TO(test-output/received-file.dat) [1 ms]
+⊗ SET_PROPERTY(*LIST) SERVICE_LIST(NME,VAL) [0 ms]
-⊗ SEND_FILE(test-output/received-file.dat) HANDLER(InboundFile) [10 ms]
+⊗ Protocol
+⊗ Build Command ( 1 ms )
+⊗ Execute Command ( 9 ms )
+⊗ SERVICE_CLOSE [21 ms]
```

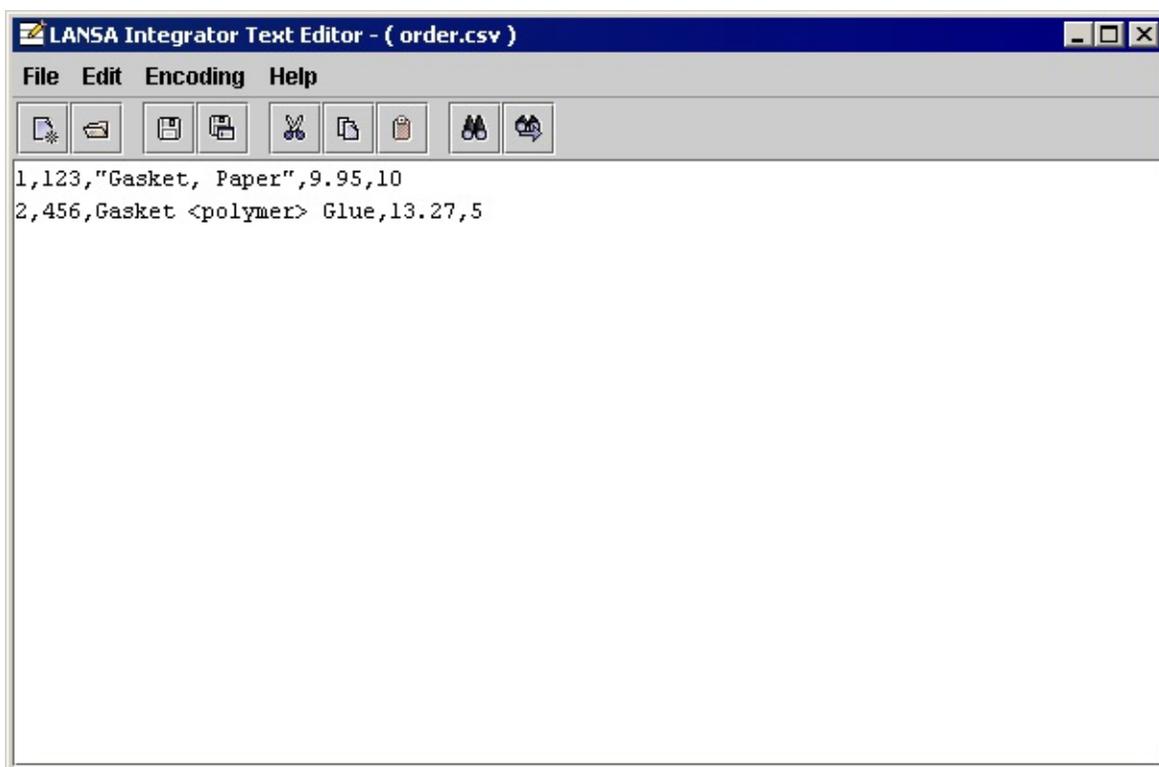
7.14 Text Editor

The Text Editor is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The Text Editor is used to maintain text files. The default selected encoding is UTF-8.

If an encoding has been selected, this value will be used to convert the byte stream to Unicode characters.

When saving to a file the selected encoding is used to convert the Unicode characters to a byte stream.

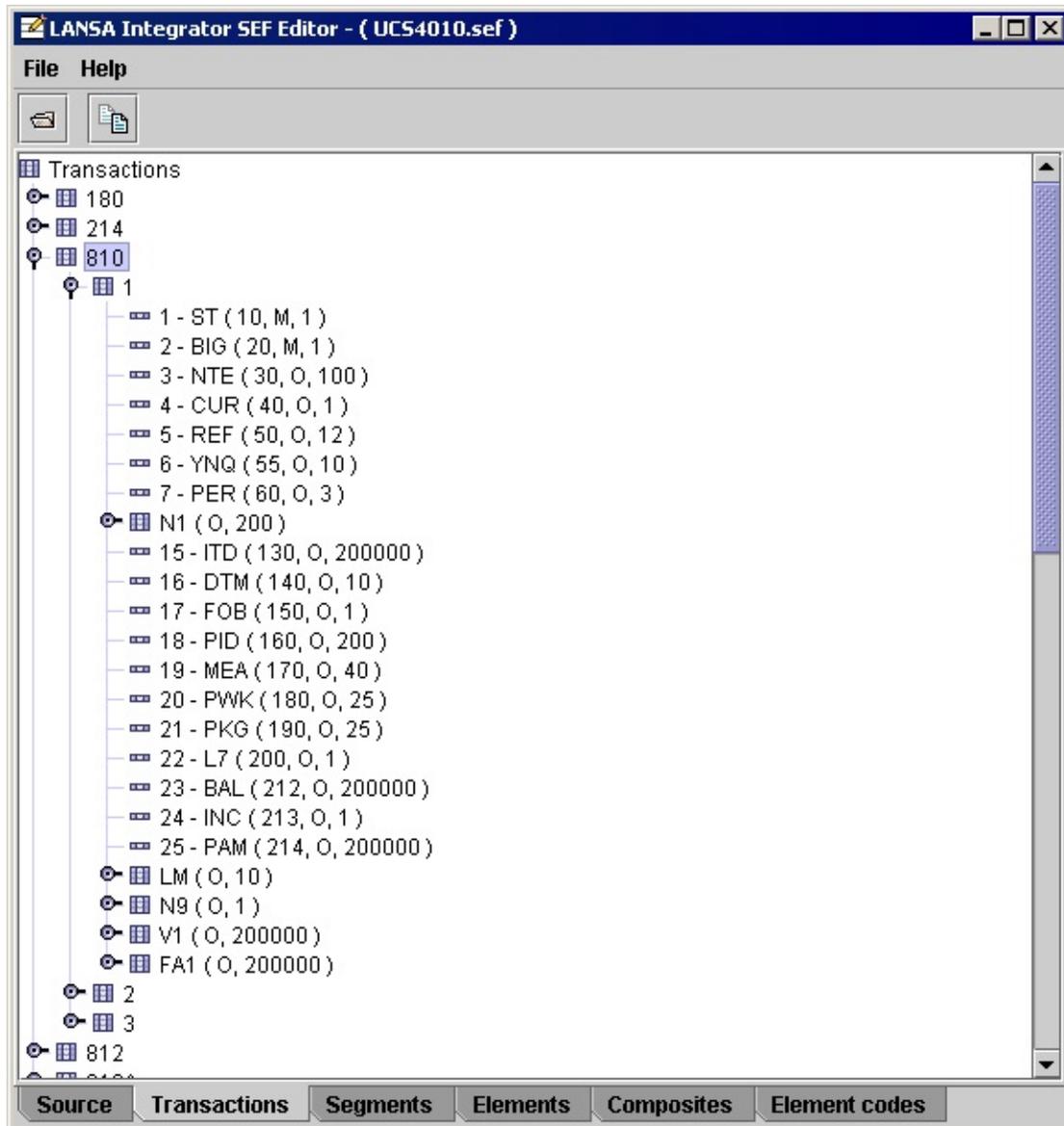


7.15 SEF Editor

The SEF Editor is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The SEF Editor is used to view EDI Standard Exchange Format (SEF) files.

The SEF Editor is associated to the file extension '.sef' and is available as an associated application from Studio.



7.16 Property Editor

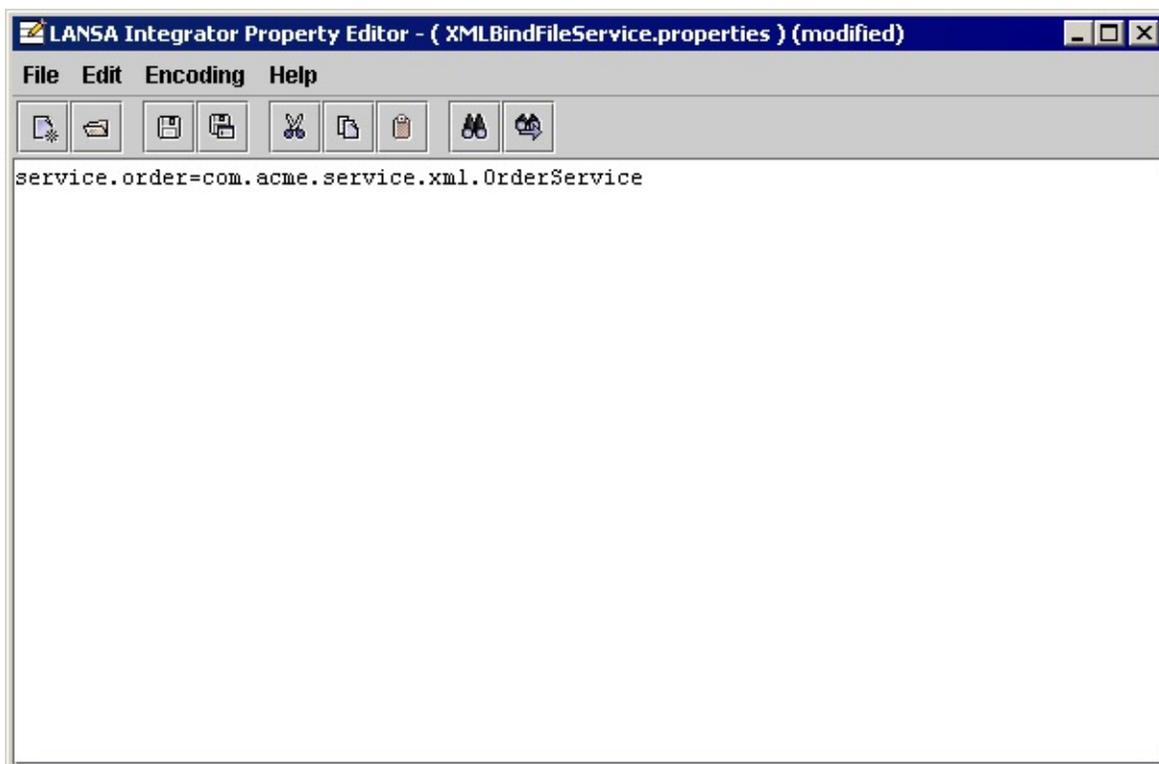
The Property Editor is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The Property Editor is used to maintain service properties files.

The encoding of service properties files is UTF-8 and this is the default selected encoding.

If an encoding has been selected, this value will be used to convert the byte stream to Unicode characters.

When saving to a file the selected encoding is used to convert the Unicode characters to a byte stream.

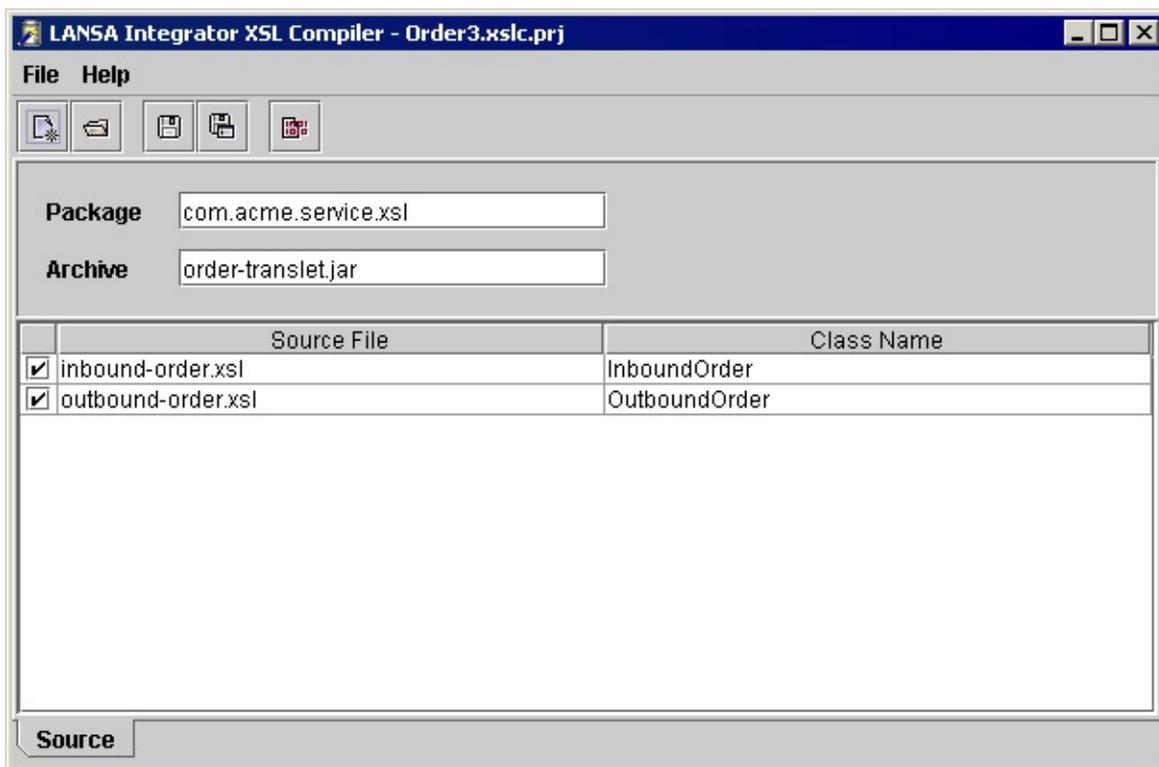


7.17 XSL Compiler

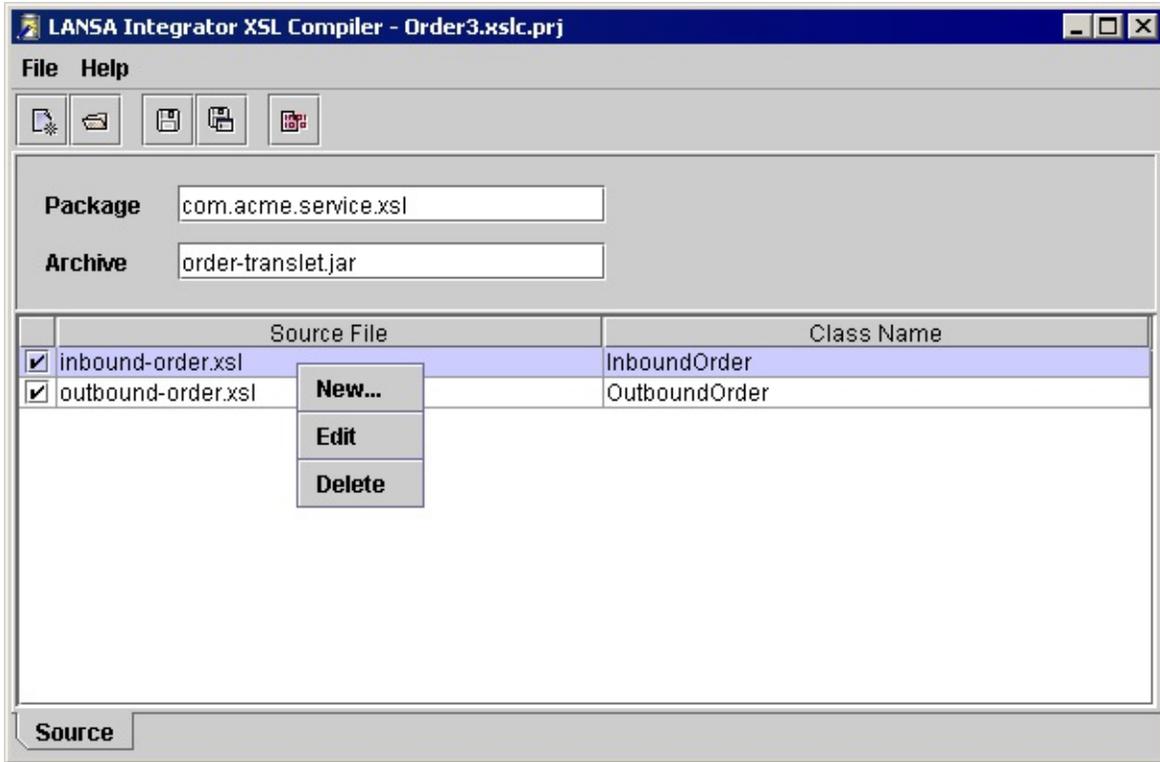
The XSL Compiler is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The XSL Compiler allows XSL files to be compiled into Java byte code classes. The XML transformers integrated into the JSM services will use the Java class instead of the XSL file to perform transformation on the XML source.

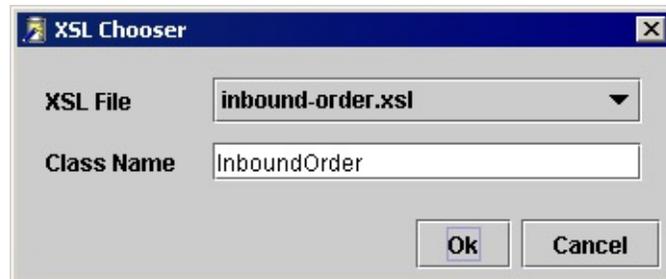
Only table entries that have been checked will be compiled and archived in to the JAR file.



To add, delete or edit an entry, right click on table to access the pop-up menu.



Only XSL files from the xsl-source sub directory are displayed in the XSL File combo.



7.18 PKI Editor

The PKI Editor is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The PKI Editor is used to create X.509 certificates, which are used by the SMTP mail service for S/MIME support and AS2 services for content encoding and decoding. For more details, refer to [7.18.1 Introduction to Certificates](#).

The PKI Editor can be used to perform the following tasks.

- [7.18.2 Create a PKI Project](#)
- [7.18.3 Create a Certificate Authority](#)
- [7.18.4 Create a Certificate Request](#)
- [7.18.5 Create a Certificate Client](#)
- [7.18.6 PKI Editor Configuration](#)
- [7.18.7 View Certificate](#)
- [7.18.8 View Certificate Request](#)
- [7.18.9 View Remote Host](#)
- [7.18.10 View Certificates and Keystores using Studio](#)
- [7.18.11 Keystore Management](#)
- [7.18.12 PEM Format](#)
- [7.18.13 IBM i Digital Certificate Manager Interoperability](#)

7.18.1 Introduction to Certificates

What is a Certificate?

A certificate is a public key labeled with information to identify its owner (Subject Name) and to control its use.

What is self-signed certificate?

A self-signed certificate is one for which the Issuer Name (signer) is the same as the Subject Name (owner).

What makes a certificate a CA certificate?

When it is used to issue other certificates.

It also contains information (extensions) that support its roles of issuing certificates (CRLDistPoint, BasicConstraints etc...)

It should also contain the BasicConstraints extension with the CA flag set to true.

Root or top-level CA certificates are self-signed.

What is the certificate thumbprint

The certificate thumbprint is a hash calculated on the whole certificate.

This thumbprint is calculated every time a certificate is displayed - it is not contained in the certificate.

What goes into a Certificate?

The X.509 standard defines what information can go into a certificate, and describes how to write it down (the data format).

All X.509 certificates have the following data, in addition to the signature:

Version	Identifies which version of the X.509 standard applies to this certificate. This affects what information can be specified in it.
Serial Number	The entity that created the certificate is responsible for assigning it a serial number to distinguish it from other certificates it issues. This information is used in numerous ways, for example when a certificate is revoked its serial number is placed in a Certificate Revocation List (CRL).
Signature Algorithm	The algorithm used by the CA to sign the certificate.

Identifier	
Issuer Name	<p>The name of the entity that signed the certificate. This is normally a CA.</p> <p>Using this certificate implies trusting the entity that signed this certificate.</p> <p>Note that in some cases, such as root or top-level CA certificates, the issuer signs its own certificate.</p>
Validity Period	<p>Each certificate is valid only for a limited amount of time. This period is described by a start date and time and an end date and time, and can be as short as a few seconds or almost as long as a century.</p> <p>The validity period chosen depends on a number of factors, such as the strength of the private key used to sign the certificate or the amount one is willing to pay for a certificate. This is the expected period that entities can rely on the public value, if the associated private key has not been compromised.</p>
Subject Name	<p>The name of the entity whose public key the certificate identifies.</p> <p>This name uses the X.500 standard, so it is intended to be unique across the Internet.</p> <p>This is the Distinguished Name (DN) of the entity, for example, CN=Road Runner, OU=Rocket Powered Systems, O=Acme Corporation, C=AU (These refer to the subject's Common Name, Organizational Unit, Organization, and Country.)</p>
Subject Public Key	<p>The public key of the entity being named, together with an algorithm identifier that specifies which public key crypto system this key belongs to and any associated key parameters.</p>

Versions

Version 1 has been available since 1988, is widely deployed, and is the most generic.

Version 2 introduced the concept of subject and issuer unique identifiers to

handle the possibility of reuse of subject and/or issuer names over time. Most certificate profile documents strongly recommend that names not be reused, and that certificates should not make use of unique identifiers. Version 2 certificates are not widely used.

Version 3 is the most recent (1996) and supports the notion of extensions, whereby anyone can define an extension and include it in the certificate.

Some common extensions in use today are:

- KeyUsage (limits the use of the keys to particular purposes such as "signing-only")
- AlternativeNames (allows other identities to also be associated with this public key, e.g. DNS names, email addresses, IP addresses)

Extensions can be marked critical to indicate that the extension should be checked and enforced/used.

For example, if a certificate has the KeyUsage extension marked critical and set to "keyCertSign" then if this certificate is presented during SSL communication, it should be rejected, as the certificate extension indicates that the associated private key should only be used for signing certificates and not for SSL use.

Encoding Format

All the data in a certificate is encoded using two related standards called ASN.1/DER. Abstract Syntax Notation 1 describes data. The Definite Encoding Rules describe a single way to store and transfer that data.

In the X.500, X.509 and X.520 standards the structures are specified with Abstract Syntax Notation 1 (ASN.1) and are encoded for transport using the Basic Encoding Rules (BER) that encode ASN.1 as 8-bit binary data.

In addition, when they represent data to be signed, as they do within certificate management, the signature is calculated on the data encoded using Distinguished Encoding Rules (DER: a subset of BER which has the property that the same data always encodes to the same binary representation).

Information stored in a certificate is a sequence of ASN.1 Objects each labeled with an object Identifier (OID).

An object identifier is a string of numbers identifying a unique object, for example, a certificate extension or a company's certificate practice statement.

OIDs are controlled by the International Standards Organization (ISO) registration authority.

In some cases, this authority is delegated by ISO to regional registration

authorities.

The OID is a unique sequence of hierarchical numbers in a dot notation.

Top level OID assignments:

- 0 ITU-T assigned
- 1 ISO assigned
- 2 Joint ISO/ITU-T assignment

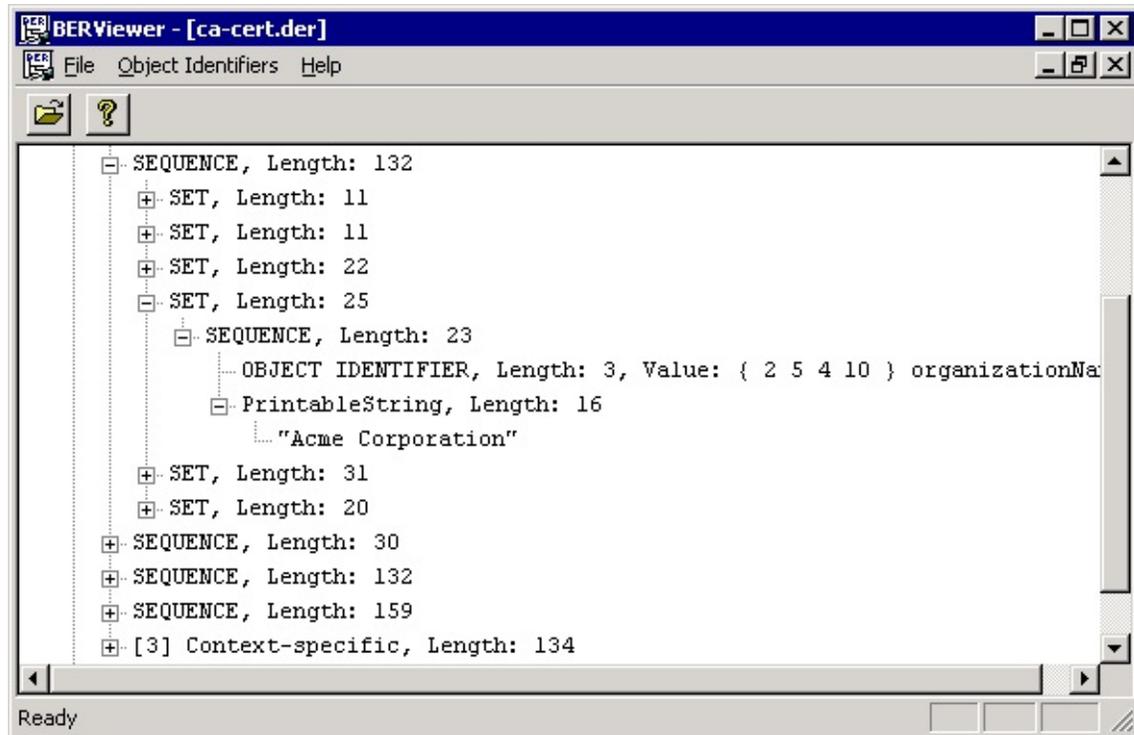
Secondary level assignments:

- 2.5 X.500 Directory Services

Other level assignments:

- 2.5.4 Object Identifiers for X.500 attributes type
 - 2.5.4.3 Common Name
 - 2.5.4.5 Serial Number
 - 2.5.4.6 Country Name
 - 2.5.4.7 Locality
 - 2.5.4.8 State
 - 2.5.4.10 Organization
 - 2.5.4.11 Organizational Unit
- 2.5.29 Object Identifiers for Version 3 extensions
 - 2.5.29.14 Subject Key Identifier
 - 2.5.29.15 Key Usage
 - 2.5.29.17 Subject Alternative Name
 - 2.5.29.19 Basic Constraints
 - 2.5.29.35 Authority Key Identifier

Certificate viewed using a generic BER viewer:



Critical and Non-critical extensions

If an extension is critical it can only be used for the purposes indicated.

If an extension is non-critical it is an advisory field and not restrictive.

Standard CA Key Usage Dialog: Standard Certificate Key Usage Dialog:



Key Usage

The critical key usage extension controls how the public key can be used.

RFC 2459 Internet X.509 Public Key Infrastructure Certificate and CRL Profile describe the role of the different key usage extensions bits.

CRL Sign is enabled when the public key is used for verifying a signature on a CRL. Enable for CA certificates.

Data Encipherment is enabled when the public key is used for enciphering user data, other than cryptographic keys.

Decipher Only and **Key Agreement** are enabled, when the public key is being used only for deciphering data while performing key agreement.

Digital Signature is enabled when the public key is used with a digital signature mechanism to support security services other than non-repudiation, key certificate signing, or CRL signing. Enable for SSL client certificates and S/MIME signing certificates.

Encipher Only and **Key Agreement** are enabled, when the public key is being used only for enciphering data while performing key agreement.

Key Agreement is enabled when the public key is used for key agreement.

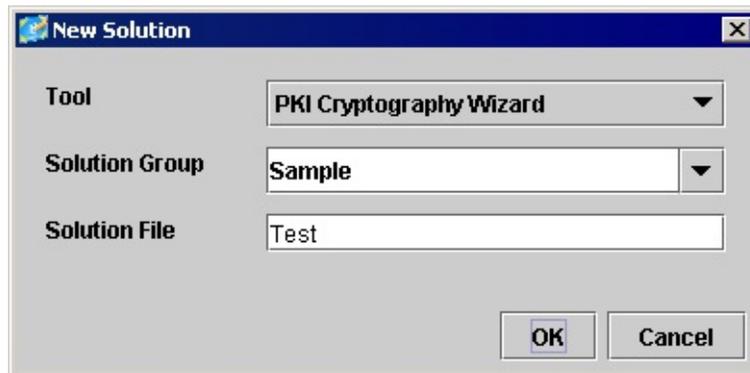
Key Certificate Sign is enabled when the public key is used for verifying a signature on certificates. Enable for CA certificates.

Key Encipherment is enabled when the public key is used for key transport. Enable for SSL server certificates and S/MIME encryption certificates.

Non Repudiation is enabled when the public key is used to verify digital signatures. Enable for S/MIME signing certificates and object-signing certificates.

7.18.2 Create a PKI Project

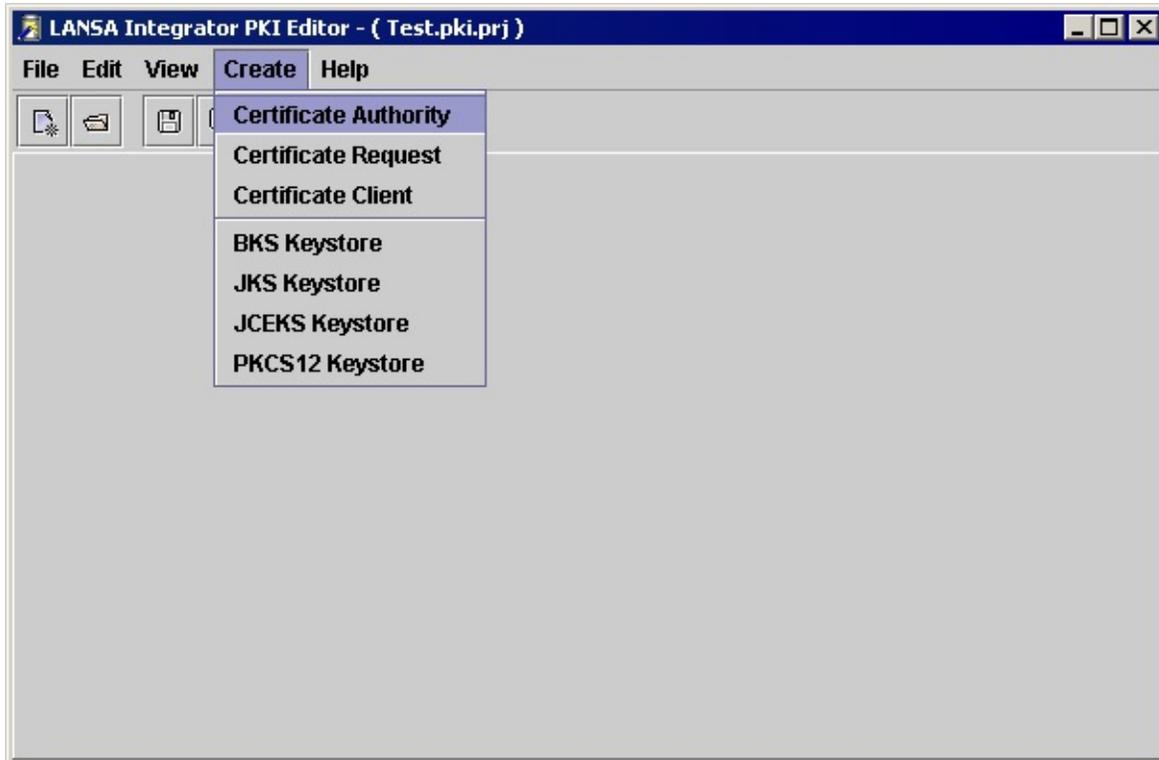
1. From the Project Solutions node select the *New Solution* menu item.
2. Select the *PKI Cryptography Wizard* tool and enter or select the *Group* folder to receive the project file.



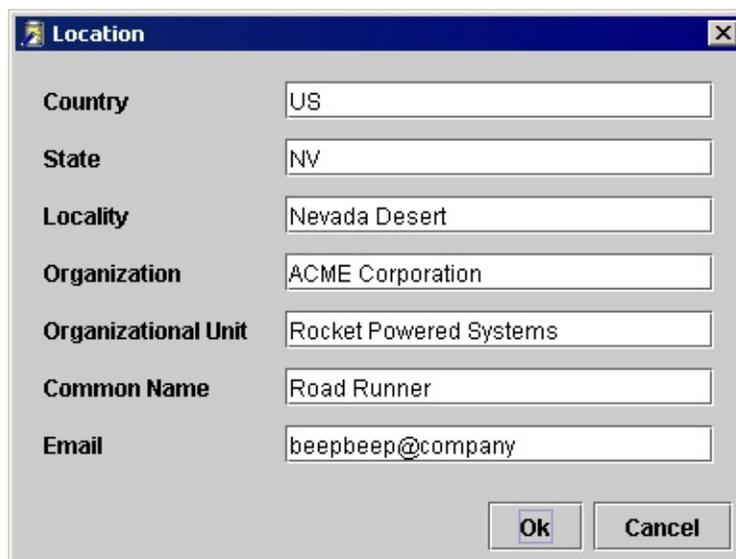
3. Enter the project File name. The extension '.pki.prj' is added automatically.

7.18.3 Create a Certificate Authority

1. Select *Certificate Authority* from the *Create* menu:



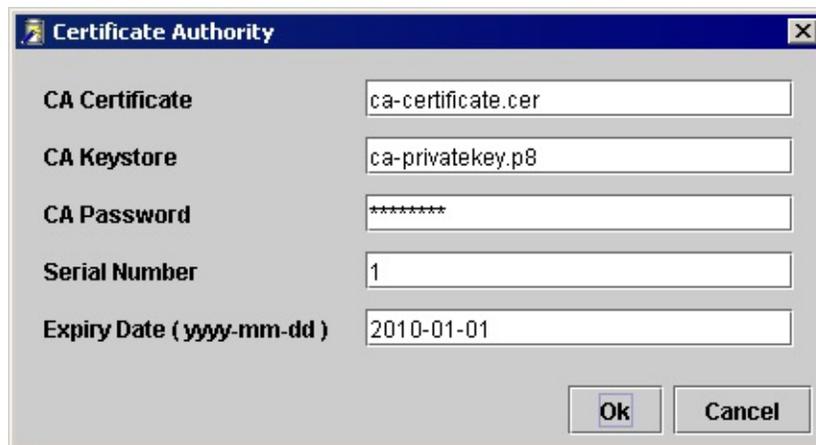
2. Enter the subject name information. The subject name information is also used for the issuer name.

A screenshot of the "Location" dialog box. It contains several text input fields with the following values: Country: US, State: NV, Locality: Nevada Desert, Organization: ACME Corporation, Organizational Unit: Rocket Powered Systems, Common Name: Road Runner, and Email: beepbeep@company. At the bottom right, there are "Ok" and "Cancel" buttons.

3. Select the key usage:



4. Enter the certificate file, private key file, password, serial number and expiry date:



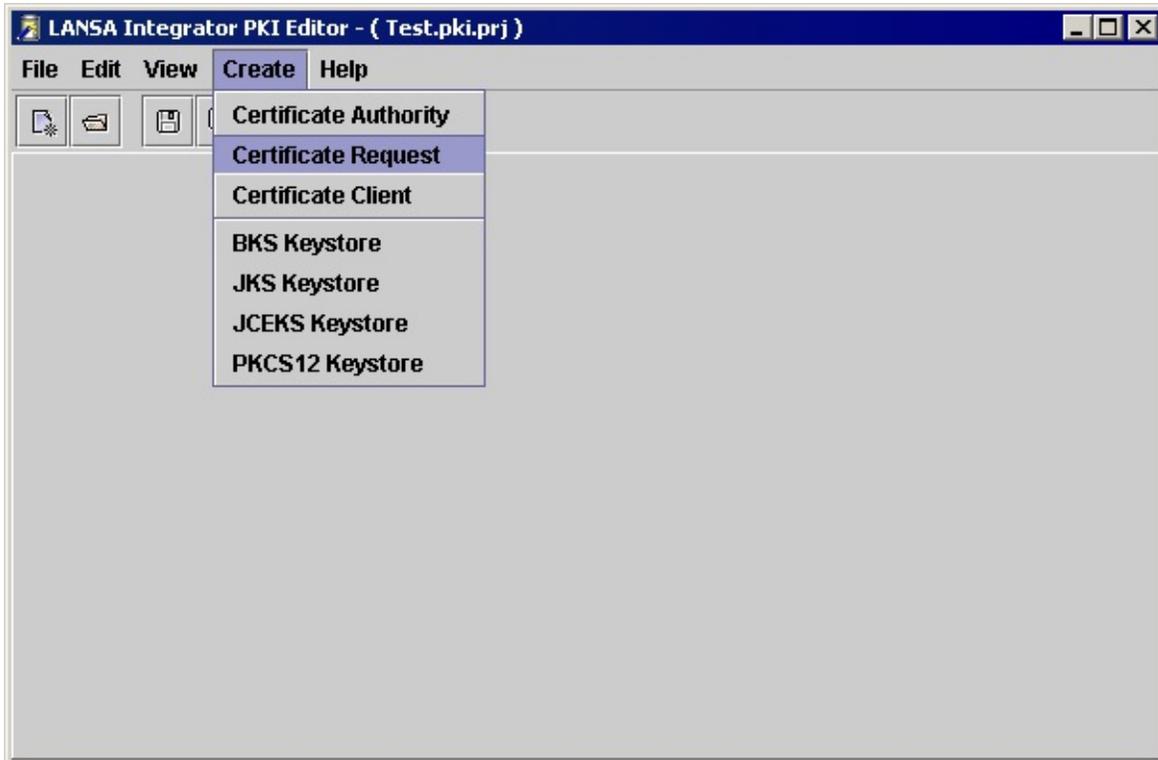
Create Certificate Authority processing steps

1. Create private and public key (save private key in PKCS#8 format)
2. Create X500 Name (X509 certificate subject)
3. Set valid date range
4. Set serial number
5. Set basic constraint to CA, unlimited path length and flag critical

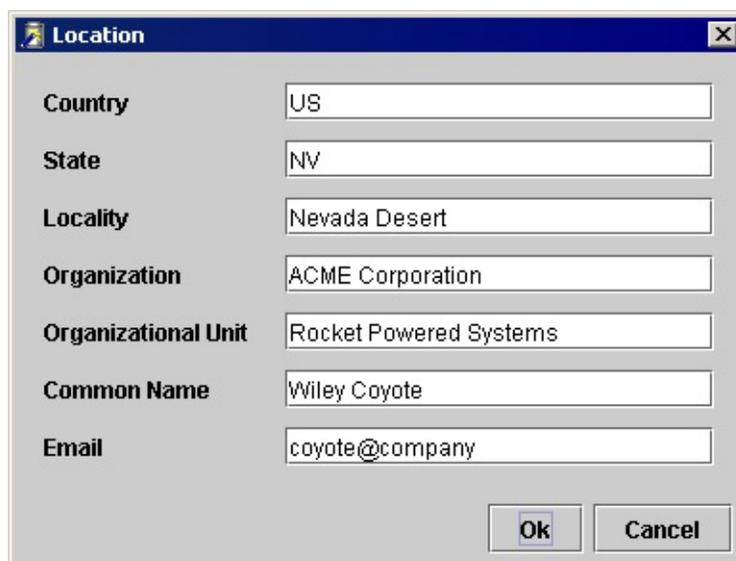
6. Set key usage extension and flag critical
7. Set subject alternate name extension (email)
8. Set subject key id extension
9. Set authority key id extension
10. Sign and save X509 certificate (This is self-signed, subject and issuer are the same).

7.18.4 Create a Certificate Request

1. Select *Certificate Request* from the *Create* menu:



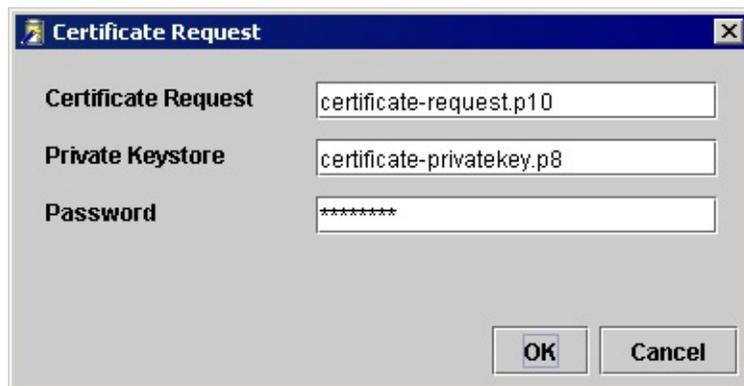
2. Enter the subject name for the certificate:

A screenshot of the "Location" dialog box. It contains several text input fields with the following values: Country: US, State: NV, Locality: Nevada Desert, Organization: ACME Corporation, Organizational Unit: Rocket Powered Systems, Common Name: Wiley Coyote, and Email: coyote@company. At the bottom right, there are "Ok" and "Cancel" buttons.

3. Select the key usage for this certificate:



4. Enter the certificate request file, private key file and password:



Create Certificate Request processing steps

1. Create private and public key (save private key PKCS#8 format)
2. Create X500 Name (X509 certificate subject)
3. Create certificate request
4. Set basic constraint and flag critical
5. Set key usage extension and flag critical
6. Set subject alternate name extension (email)
7. Set subject key id extension

8. Sign and save request (PKCS#10 format).

The request contains the subject's name and public key, and is signed with the subject's private key.

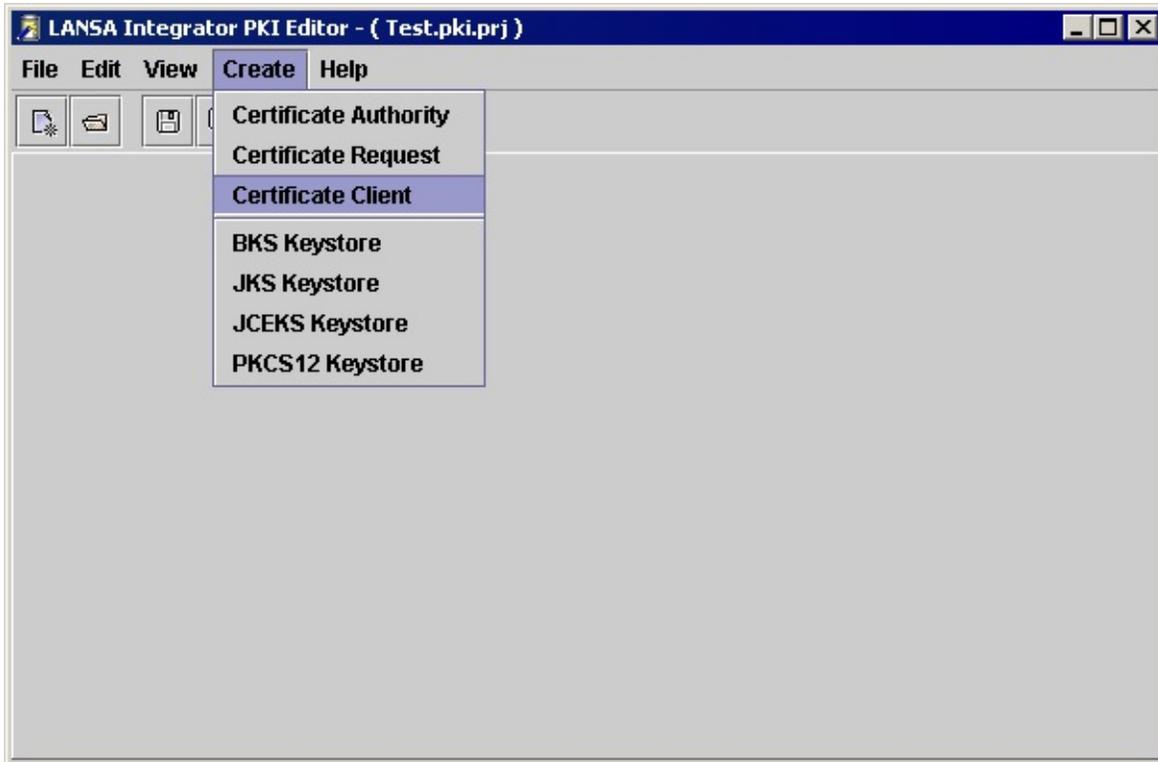
Note that the subject's private key is used only to produce a signature when the request is output, and is not actually stored with the request.

The file containing the certificate request (DER encoded) can be converted to PEM format and sent to a CA authority. The CA authority will use the certificate request to create a certificate signed with its public certificate.

Or you can create your own certificate. You do not need to convert the file for this stage.

7.18.5 Create a Certificate Client

1. Select *Certificate Client* from the *Create* menu:



2. In the Certificate Client dialog, enter CA certificate file, CA private key file, password, certificate request file, serial number, number of days for which the certificate is valid and the file name of the client certificate to be created.

A screenshot of the "Certificate Client" dialog box. It contains several input fields and two buttons at the bottom. The fields are: "CA Certificate" with the value "ca-certificate.cer"; "CA Keystore" with the value "ca-privatekey.p8"; "CA Password" with the value "*****"; "Certificate Request" with the value "certificate-request.p10"; "Certificate Client" with the value "certificate.cer"; "Serial Number" with the value "2"; and "Valid Days" with the value "365". The "Ok" and "Cancel" buttons are located at the bottom right of the dialog.

Create Certificate Client processing steps

1. Read CA private key and certificate
2. Read certificate request (X509 certificate subject)
3. Create certificate
4. Set valid days
5. Set serial number
6. Sign and save X509 certificate (signed by CA).

7.18.6 PKI Editor Configuration

Select the encryption algorithm and key size

Your selection really depends on your application:

- How long does the data need to be secure?
- How much does it cost and how much is it worth?

Recent standards currently recommend RSA key sizes of 1024 bits for corporate use and 2048 bits for extremely valuable keys like the root key pair used by a certifying authority.

Longer key sizes are more secure but this increased security comes at the cost of performance.

A doubling of the RSA module increases processing time requirements by a factor of 4 (public key operations - Signature Verification, Encryption) and 8 (private key operations - Signature Generation, Decryption).

Set the encryption algorithm and key size

Edit the project file and modify the algorithm and strength properties.

```
strength=1024
algorithm=*RSA | *DSA
certificate.signing.algorithm=*MD5RSA | *SHA1RSA | *SHA1DSA
```

When the algorithm is *RSA, choose certificate signing algorithm *MD5RSA or *SHA1RSA.

When the algorithm is *DSA, choose certificate signing algorithm *SHA1DSA.

The default key algorithm is *RSA.

The default signature algorithm is *SHA1RSA or *SHA1DSA depending on the key algorithm.

Set extended key usage

Additional key usage extensions can be added to the certificate request and client certificate by including 'extended.purpose' properties. A maximum of 20 properties can be included, starting from the sequence number of 1 and ending with the number 20.

To add the following extended key usages:

- Server Authentication (1.3.6.1.5.5.7.3.1)

- Client Authentication (1.3.6.1.5.5.7.3.2)
- Code Signing (1.3.6.1.5.5.7.3.3)
- Secure Email (1.3.6.1.5.5.7.3.4)
- Time Stamping (1.3.6.1.5.5.7.3.8)
- OCSP Signing (1.3.6.1.5.5.7.3.9)

extended.purpose.1=1.3.6.1.5.5.7.3.1

extended.purpose.2=1.3.6.1.5.5.7.3.2

extended.purpose.3=1.3.6.1.5.5.7.3.3

extended.purpose.4=1.3.6.1.5.5.7.3.4

extended.purpose.5=1.3.6.1.5.5.7.3.8

extended.purpose.6=1.3.6.1.5.5.7.3.9

Set CRL distribution

A CRL distribution extension can be included with each certificate.

crl.distribution=http://www.mycompany.com/CRLList.crl

crl.distribution=http://www.mycompany.com/crllist.html

Set Subject Alternative Names for SSL authentication

A list of SSL authentication Subject Alternative Names can be included with each certificate.

As part of the SSL trust process an SSL client program can compare the connection domain host with the domains listed in the subject alternative fields of the received SSL certificate.

Use the 'ssl.addresses' property to specify a list of IP addresses.

Use the 'ssl.domains' property to specify a list of host domain names.

ssl.addresses=10.2.0.173,10.2.0.174

ssl.domains=*.mycompany.com,support.mycompany.com,account.mycompany

Example PKI Editor project file

#JSFPKIEditor last values

#Sun Nov 02 22:34:20 GMT 2003

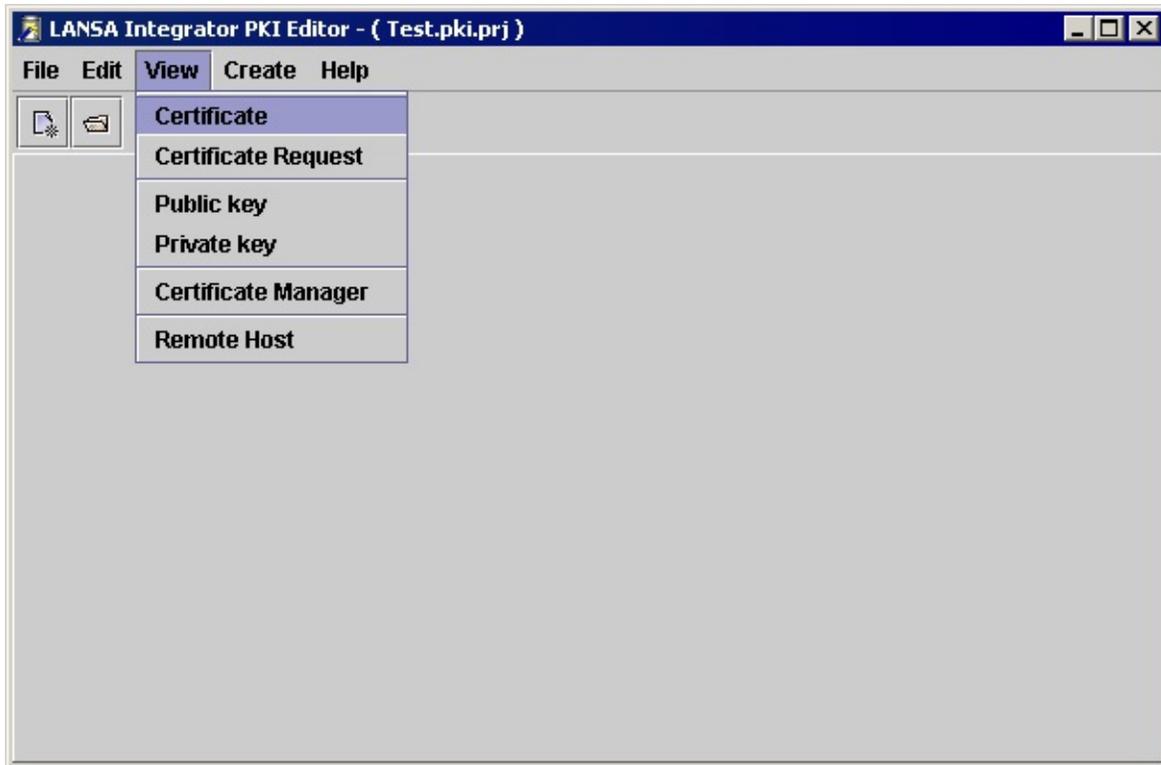
ca.keystore=ca-key.der
ca.keystore.password=
ca.certificate=ca-cert.der
ca.expiry=1/1/2005
request.keystore=request-key.der
request.keystore.password=
request.certificate=request-cert.der
certificate=certificate.der
blank.password=*yes
algorithm=*RSA
strength=1024
certificate.signing.algorithm=*SHA1RSA
serial=75
days=365
location.organization=ACME Corporation
location.unit=Rocket Powered Systems
location.locality=Nevada Desert
location.state=NV
location.country=US
location.name=Road Runner
location.email=beepbeep@acme.com
extended.purpose.1=1.3.6.1.5.5.7.3.2
extended.purpose.2=1.3.6.1.5.5.7.3.1

You need to use a text editor to set the following properties

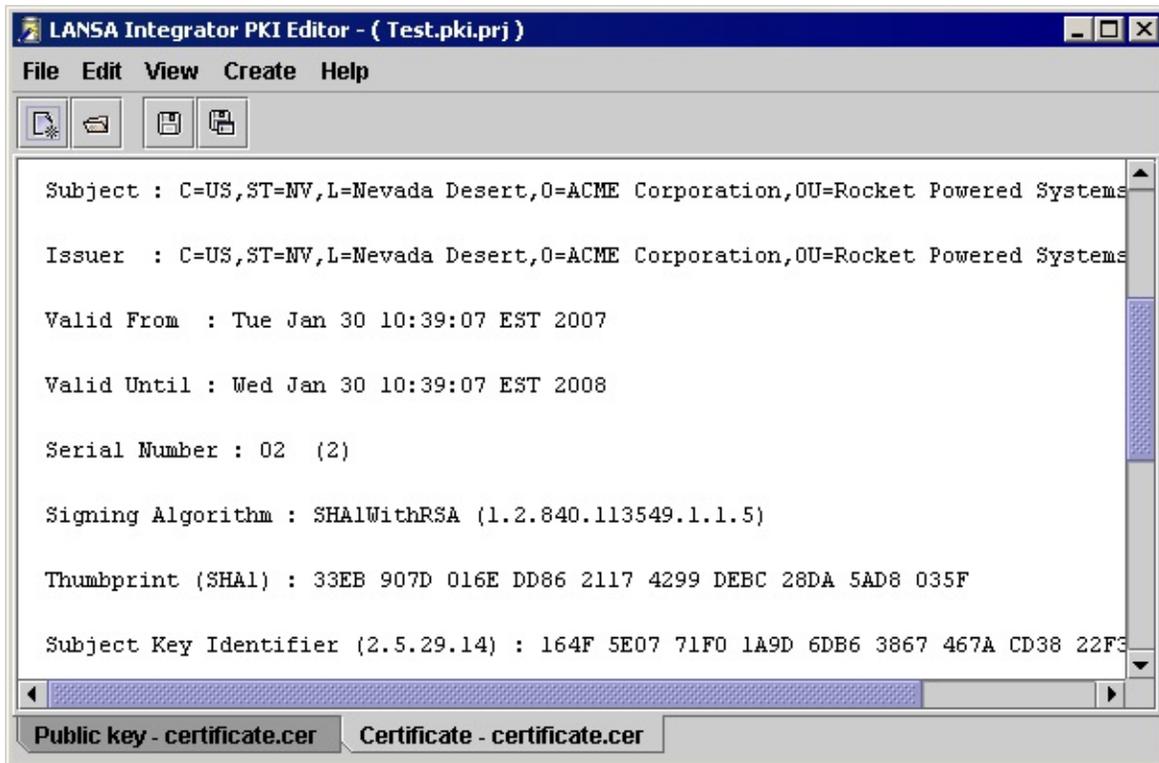
strength=1024
algorithm=*RSA | *DSA
certificate.signing.algorithm=*MD5RSA | *SHA1RSA | *SHA1DSA
blank.password=*YES | *NO

7.18.7 View Certificate

Select *Certificate* from the *View* menu:



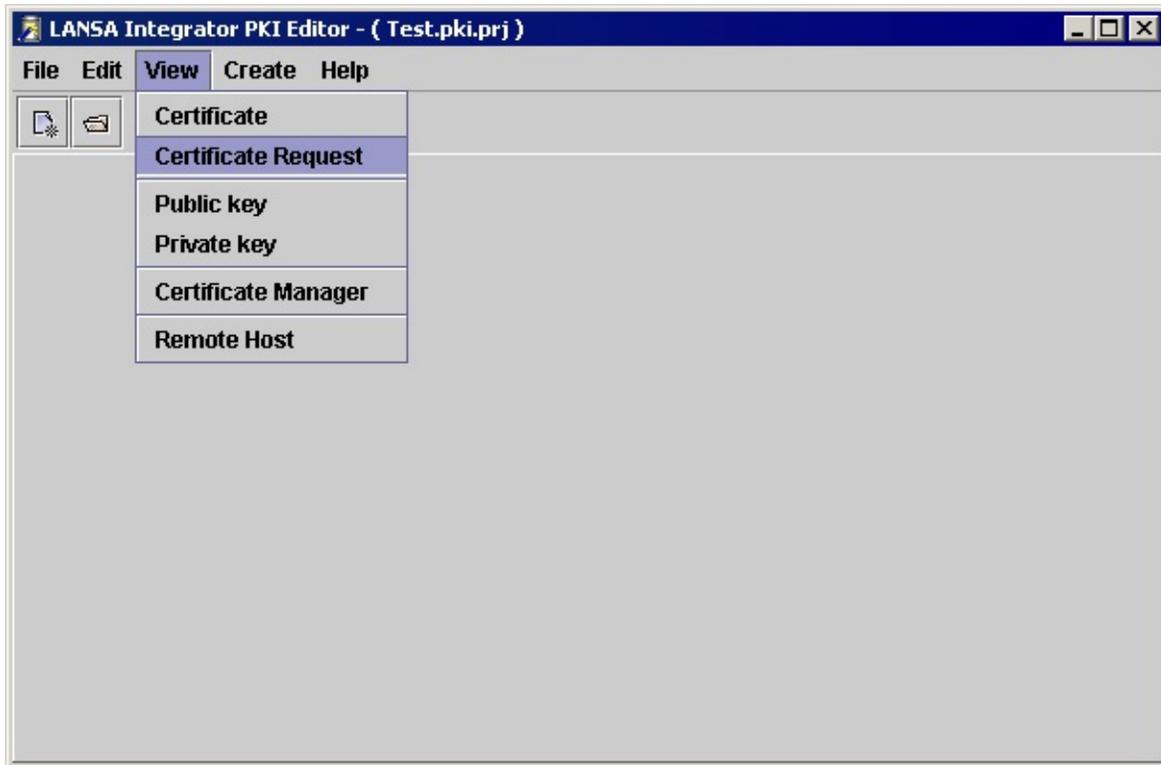
Select the DER or PEM encoded certificate file:



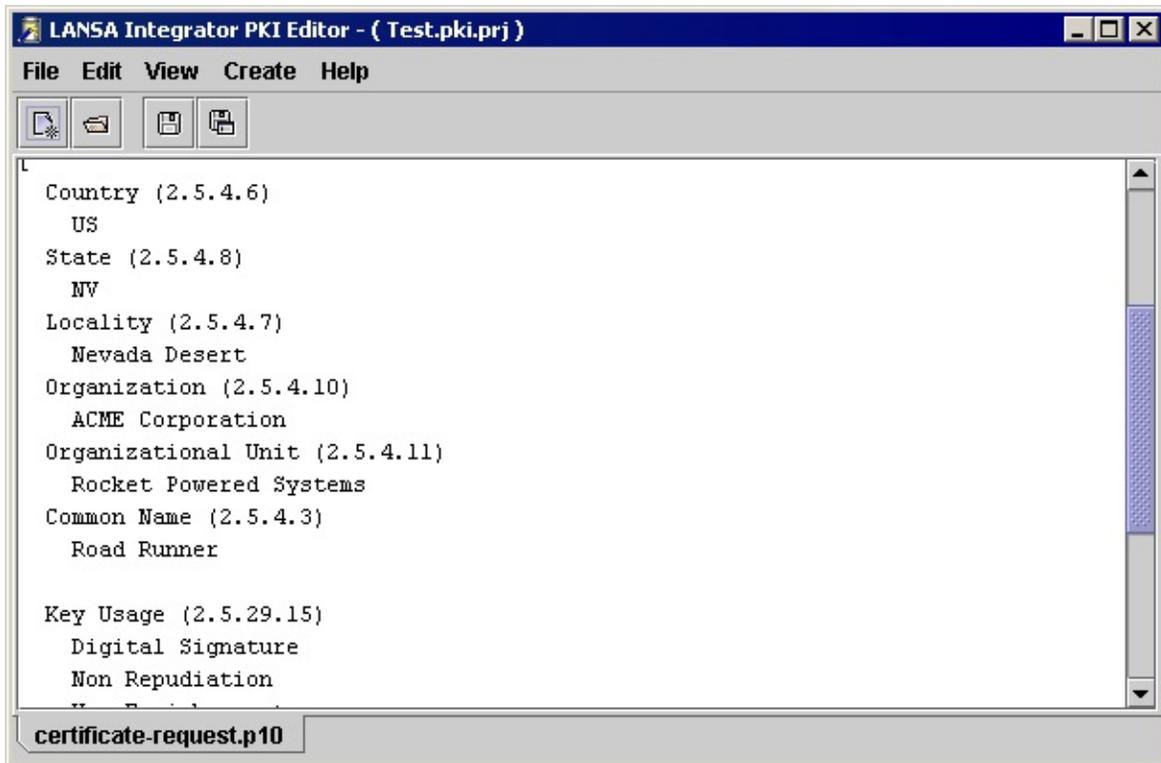
The certificate can be saved in DER or PEM format using the Save command on the pop-up menu.

7.18.8 View Certificate Request

Select *Certificate Request* from the *View* menu:



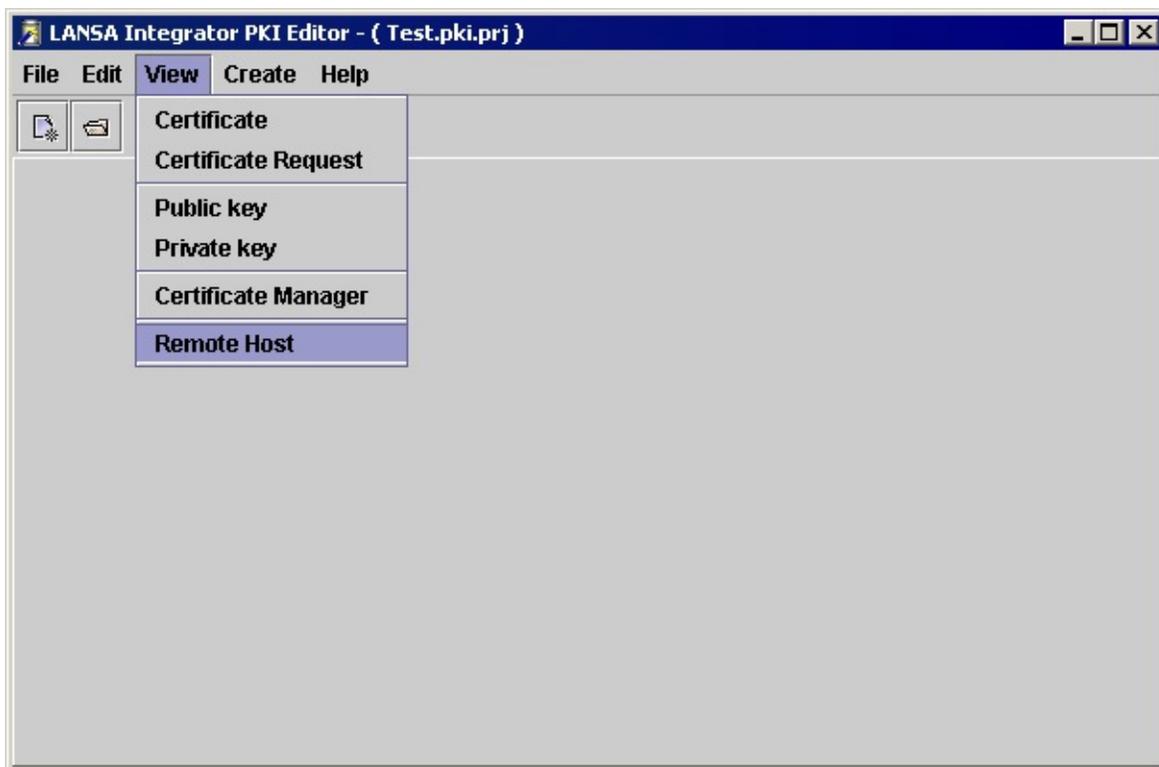
Select the DER or PEM encoded certificate request file:



The certificate request can be saved in DER or PEM format using the Save command on the pop-up menu.

7.18.9 View Remote Host

Select *Remote Host* from the *View* menu:



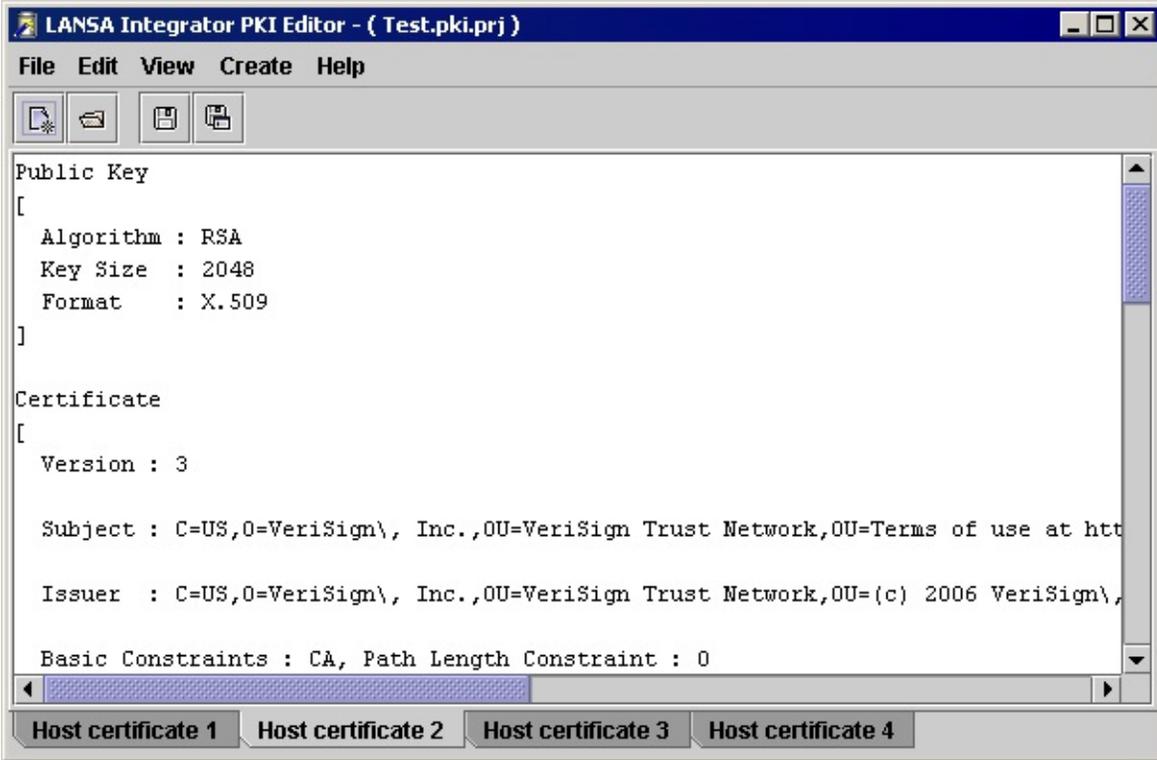
Enter the address of the SSL server.

The default port is 443.

This is protocol independent, only an SSL connection is made to get the peer certificates. Use port 990 to connect to a secure FTP server.

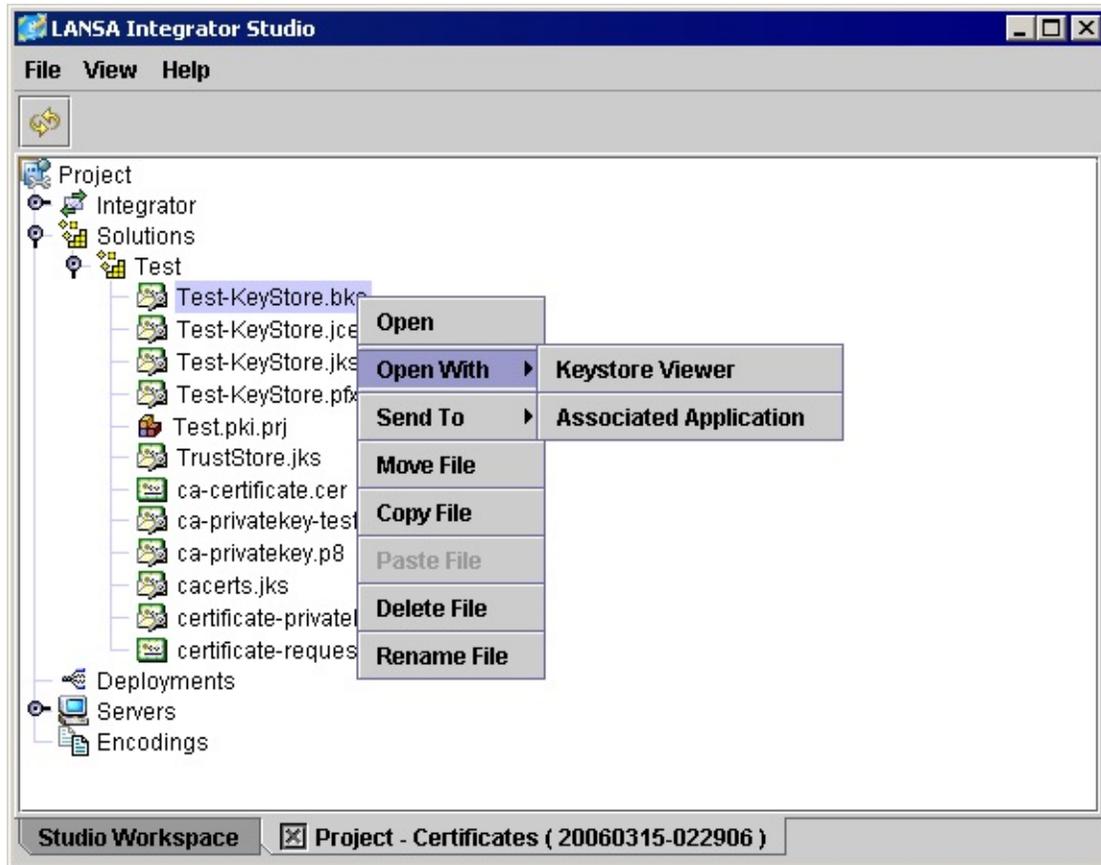


The certificate chain of the peer system is loaded into the PKI Editor. These certificates can be saved and then imported into a key store.



7.18.10 View Certificates and Keystores using Studio

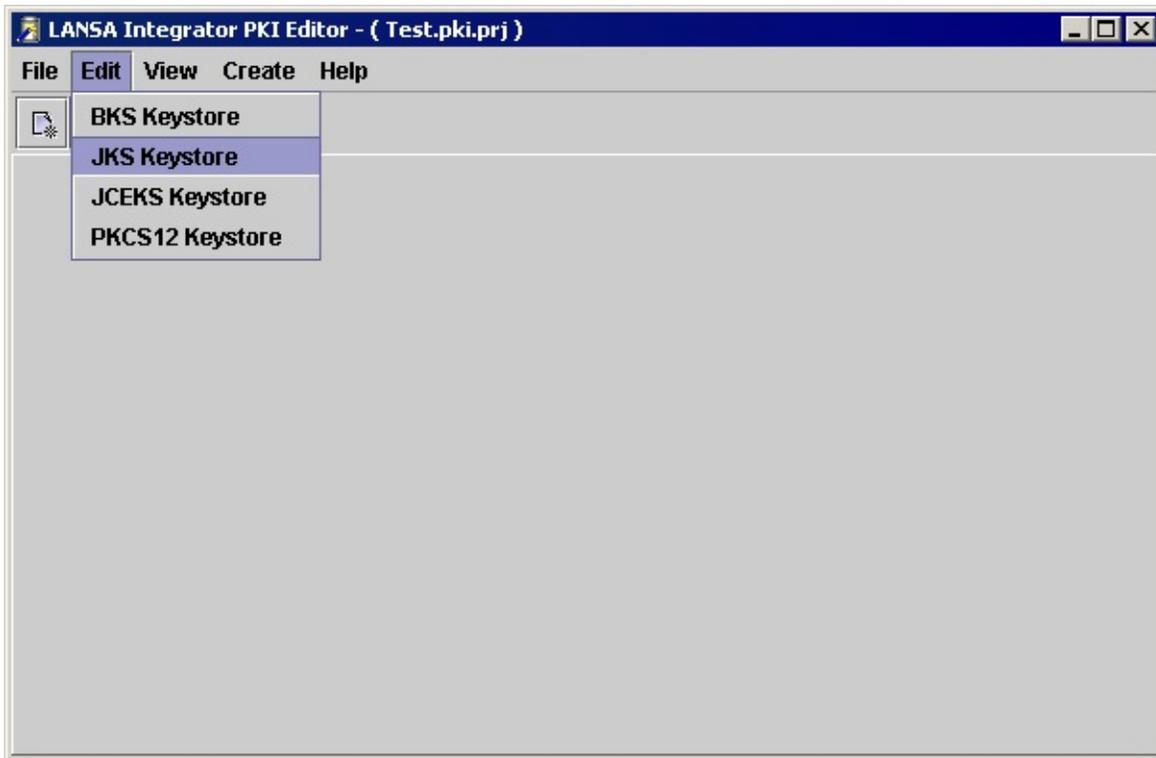
The following image illustrates that Studio can be used to view certificates and keystores.



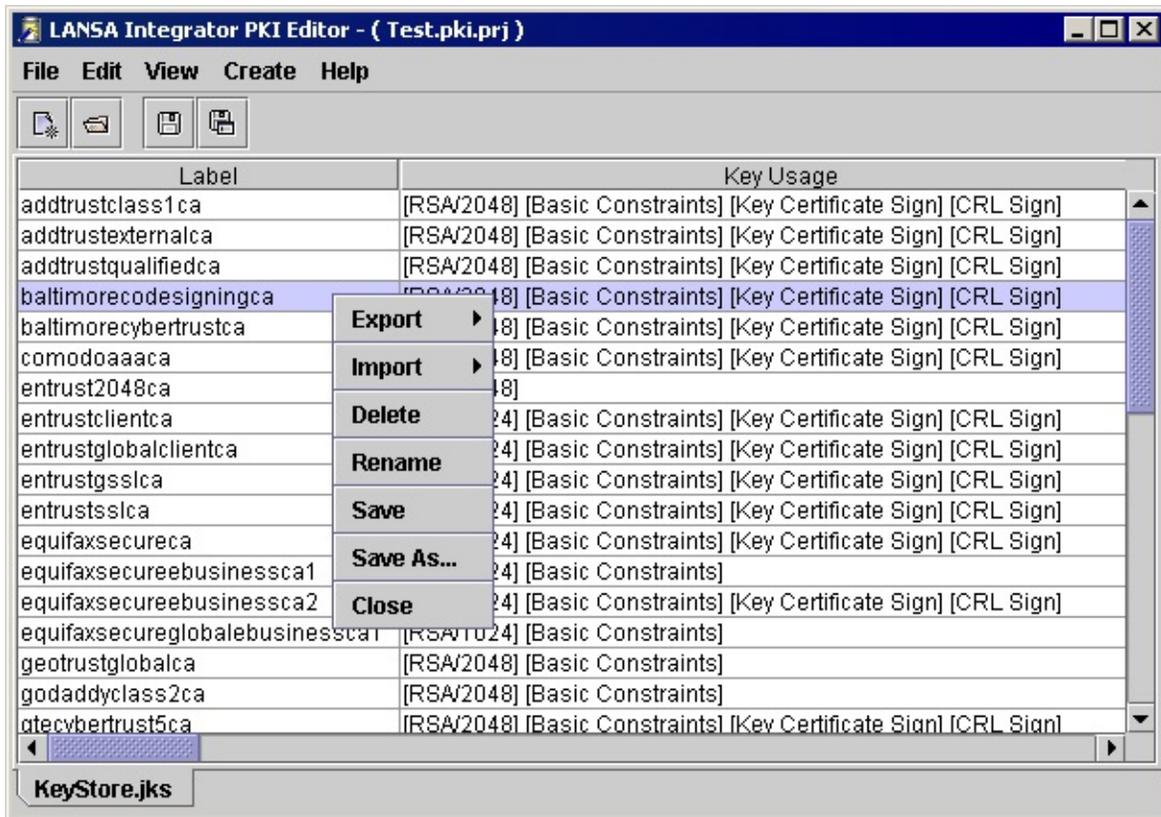
7.18.11 Keystore Management

The PKI Editor can be used to manage the contents of a key store.

Select *JKS Keystore* from the *Edit* menu:



Enter the key store file and password to open the key store:



A pop-up menu allows certificates and private keys to be imported into the key store.

If the key store is empty, right click on the table column header to select from the pop-up menu.

7.18.12 PEM Format

PEM format is a BASE64 encoding of the binary format (DER) with begin and end line markers.

X509 certificate in PEM format:

```
-----BEGIN CERTIFICATE-----
MIIDBJCCAe6gAwIBAgIISlmOxw+N3SgwDQYJKoZIhvcNAQEEBQAwwQz
BhMCQVUxDDAKBgNVBAGTA05TVzEWMBQGA1UEChMNTEFOU0Eg
A1UEAxMFTEFOU0EwHhcNMMDIwNTI1MjMxOTE4WhcNMMDUwNTMwM
CQYDVQQGEwJBVTEMMAoGA1UECBMDElNXMRYwFAYDVQQKEwV
MQ4wDAYDVQQDEwVMQU5TQTCCASIwDQYJKoZIhvcNAQEBBQAD
ALgMWPw7Lr9c5KOsyYocJoDV71VLAAX/QaH0ARXcsa/2haAxD4G5aG
aoQ9OzuOgpN2nCrXKiu7L6d1cdoxCwJ83M4akY6aZn1LhL4gL05e0SUGJ
FvK01pUUHD6lqQ0uX33lZoeGVszxsG4lc2WKniSt7QYlx0C13UmNHFn1g
FBZELF8snJyLNBXqYRWTAkSdR+xyqJMWesb+L/Uq27vjoanquipVeWlSbS
05QnBHFKM0Cxd44msmn7vDdGv0Pm9ooP6VB6IMKx0mhU9aHmae51s/q
hL3vslN9mpz2p221+06ew+ECAwEAATANBgkqhkiG9w0BAQQFAAOCAQ
2Ti2+goGeWsUoXc331aDt/I1mBBQvst8KiTbUQSMjmRTToSIE5CAG+J17Irg
TnQOBrcLimXpUhi/K4O7bbuFp9i9jsDPjPuecOKL548++2xvhhbVBzTOZN
TBWJQCQNXsoxUY0K+joC1gCH/VbRdsZfh/EnIHcpJugoWK0IhSqJHKyfVe
bwy6xnODF/QcTZ3pD5JWP2CdiuFPSg6e54rkIWihhdeuYEWjF3XveVjhvJq
DPOzaYmI17IqNcrekpPt5uzjSySTs1PJst8TBlTx0mKnWGMJVkzOHR3q7z
tCeSu9DCPBQthQ==
-----END CERTIFICATE-----
```

Certificate request in PEM format:

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIBejCB5AIBADA7MQswCQYDVQQGEwJBdTEMMAoGA1UECBMDEl
EwVMQU5TQTTEOMAwGA1UEAxMFQWxpY2swZ8wDQYJKoZIhvcNAQ
AoGBAJf9Sd2HoXVBus9hoDXv2bh5oAv7Tmz+3iJV0qYj1SGMlu0tsDmayl
t0khXs+Sk4rrZbXN7LyhKviHRxHUYd9Et98VoUoF1tSTCwa+sAoRdjWPgI
aqWr/+fphy7xDxQMFAHRjx2LDYpyoxWEdVuhW1KjaCL9khpvAgMBAAC
hkiG9w0BAQQFAAOBqBT2xVhoFGQg/IOS5IB3/QB5EYubaAwRWrqQEn
p81obId6bqrSR3HgiuwLKPeVRRUyTTrgzQCCocJzznzAlcanTEUmPoga56L
ht7ARrQPdcRiALAD7SxyXMyKPKBwyPRiDbSp0CQcxNIoyplvrLCCxin4
-----END NEW CERTIFICATE REQUEST-----
```


7.18.13 IBM i Digital Certificate Manager Interoperability

- The IBM i DCM produces certificate requests in PEM format.
- DCM can export EBCDIC encoded PEM files. Check the coded character set ID of the file before transferring to PC.
- Use FTP ASCII mode to convert the file content when transferring from the IFS to PC.
- DCM can import ASCII encoded PEM files, so transfer from the PC to the IFS using binary mode.

Create a CA certificate for the IBM i DCM

1. Use the New Certificate Authority to create a CA certificate in DER format.
2. Use the Open Certificate to read the certificate and save in PEM format.
3. FTP this PEM file to the IFS and import the CA certificate in *SYSTEM store.

Create certificate for the IBM i DCM

1. Use DCM to create a client certificate request, copy the PEM encoded certificate request shown in the browser text area to a file.
2. Use the Open Certificate Request to read the PEM encoded file and save in DER format.
3. Use the New Certificate Client to read the local CA file and certificate request file to create a certificate.
4. Use Open Certificate to read certificate and save in PEM format.
5. FTP PEM encoded certificate to the IBM i IFS and import.

7.19 CRL Editor

The CRL Editor is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

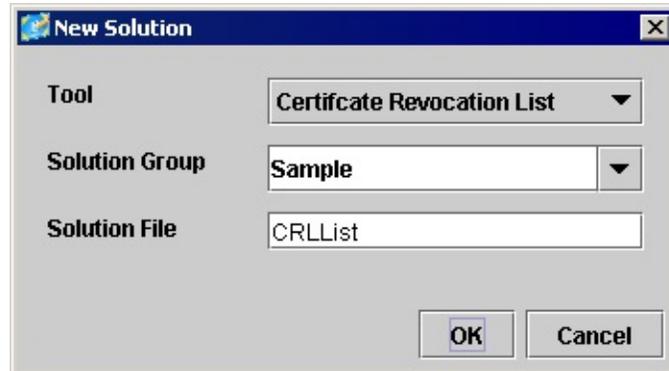
The CRL Editor is an application that allows the creation of certificate revocation list (CRL) files as described in these steps:

Review the following topics:

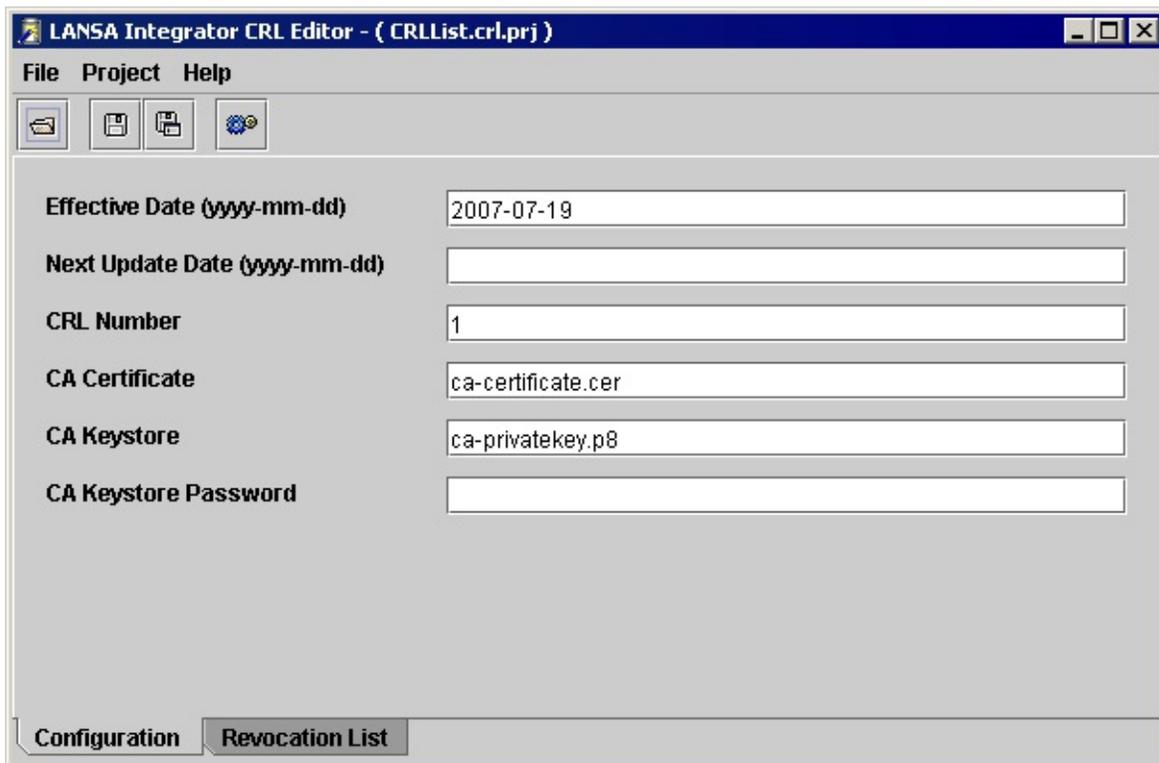
- [7.19.1 Create a CRL Project](#)
- [7.19.2 Distribute a CRL File](#)

7.19.1 Create a CRL Project

1. From the Project Solutions node select the *New Solution* menu item.
2. Select the *Certificate Revocation List* tool and enter or select the *Group* folder to receive the project file.
3. Enter the project File name. The extension '.crl.prj' is added automatically.

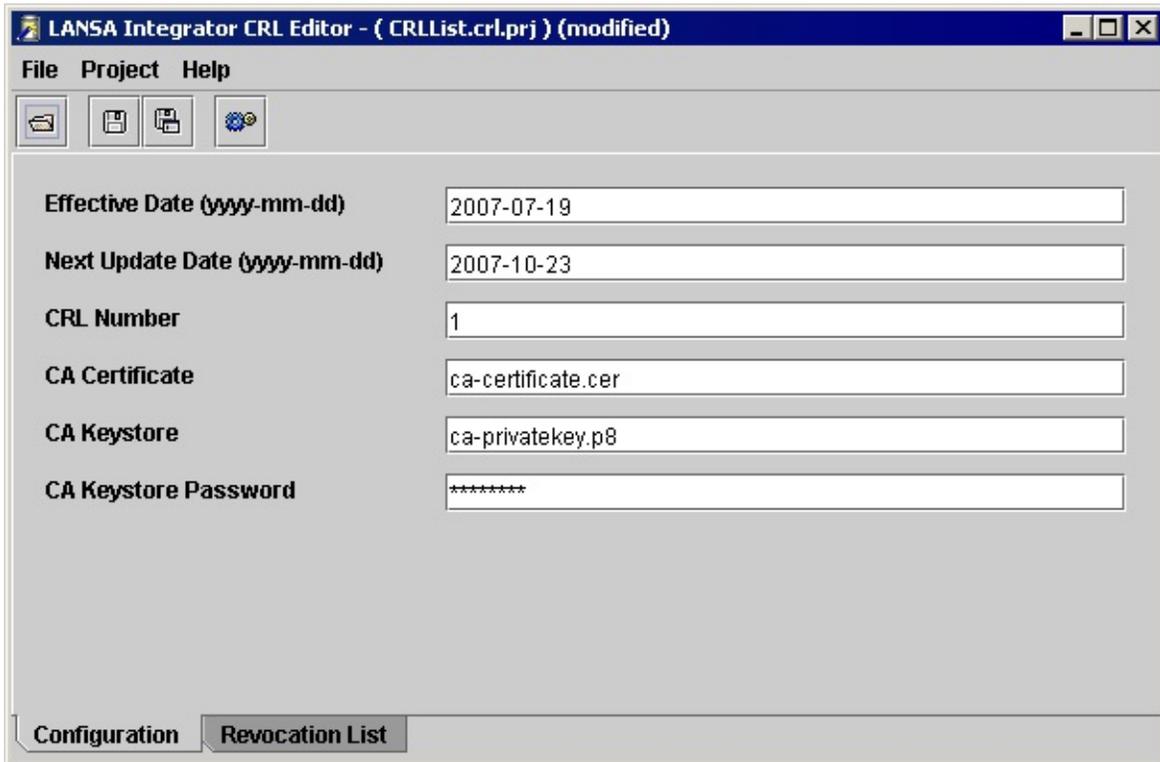


4. Press the OK button to create the project.



- a. Enter the date when the next CRL list will be available.
- b. Enter the unique CRL sequence number.

- c. Enter the file that contains the CA certificate that signed the certificates that are to be revoked.
 - d. Enter the file that contains the CA private key, so the CRL file can be signed.
5. Enter the private key password

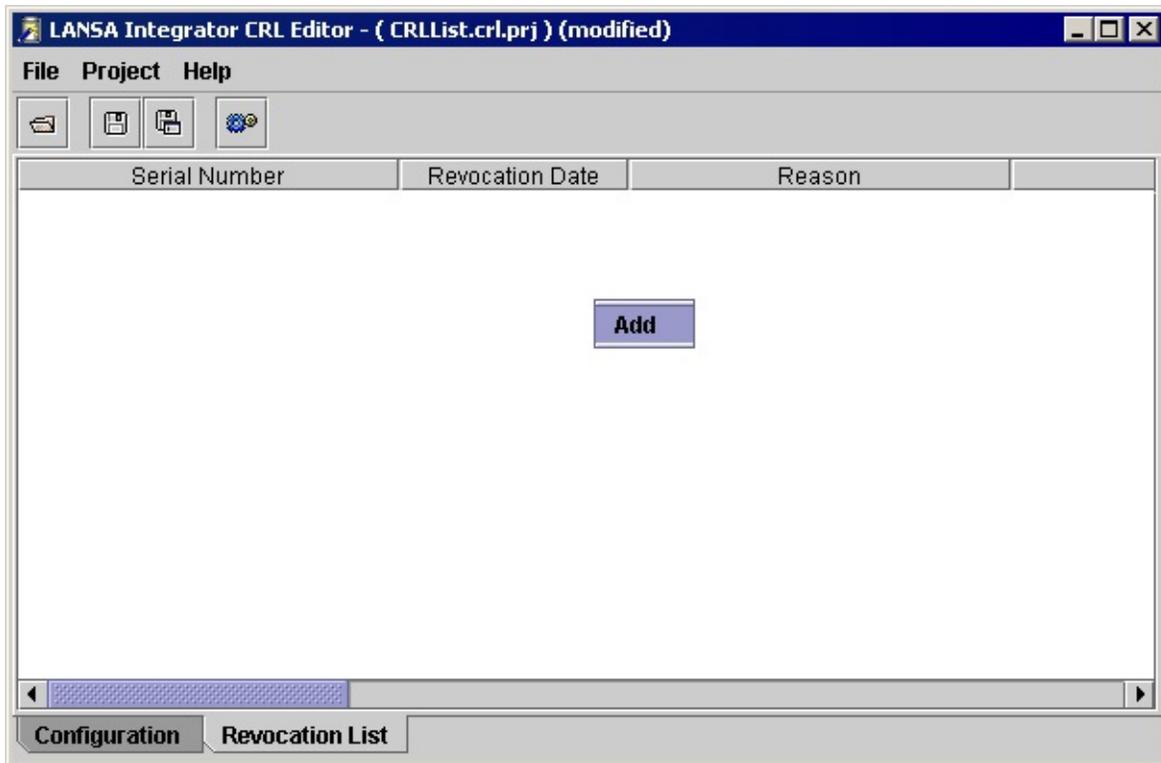


The screenshot shows the 'LANSA Integrator CRL Editor - (CRLList.crl.prj) (modified)' window. It has a menu bar with 'File', 'Project', and 'Help'. Below the menu bar are icons for file operations and settings. The main area contains several input fields:

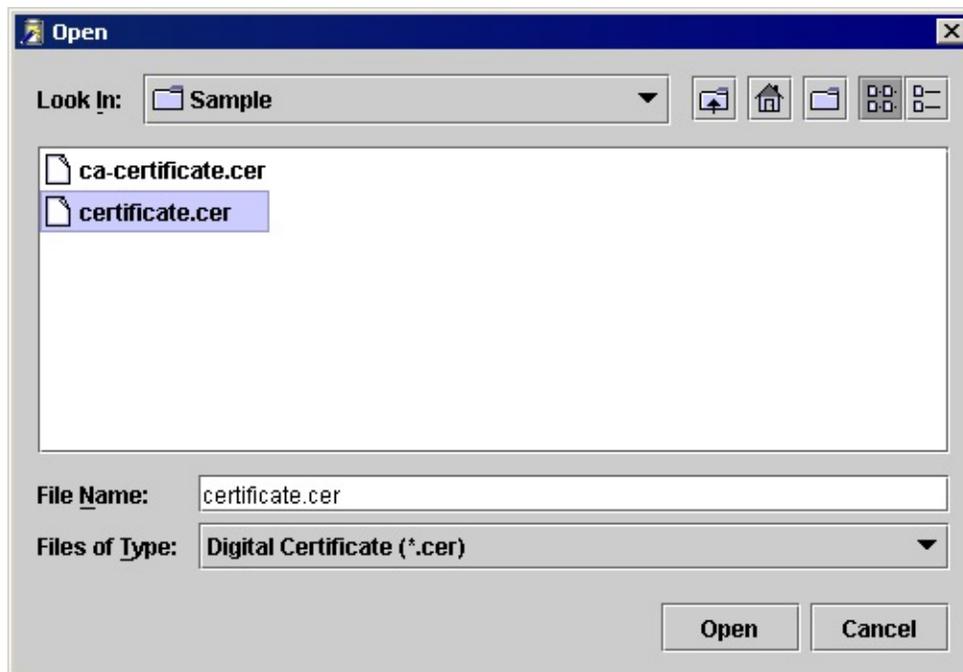
Effective Date (yyyy-mm-dd)	2007-07-19
Next Update Date (yyyy-mm-dd)	2007-10-23
CRL Number	1
CA Certificate	ca-certificate.cer
CA Keystore	ca-privatekey.p8
CA Keystore Password	*****

At the bottom, there are two tabs: 'Configuration' and 'Revocation List'.

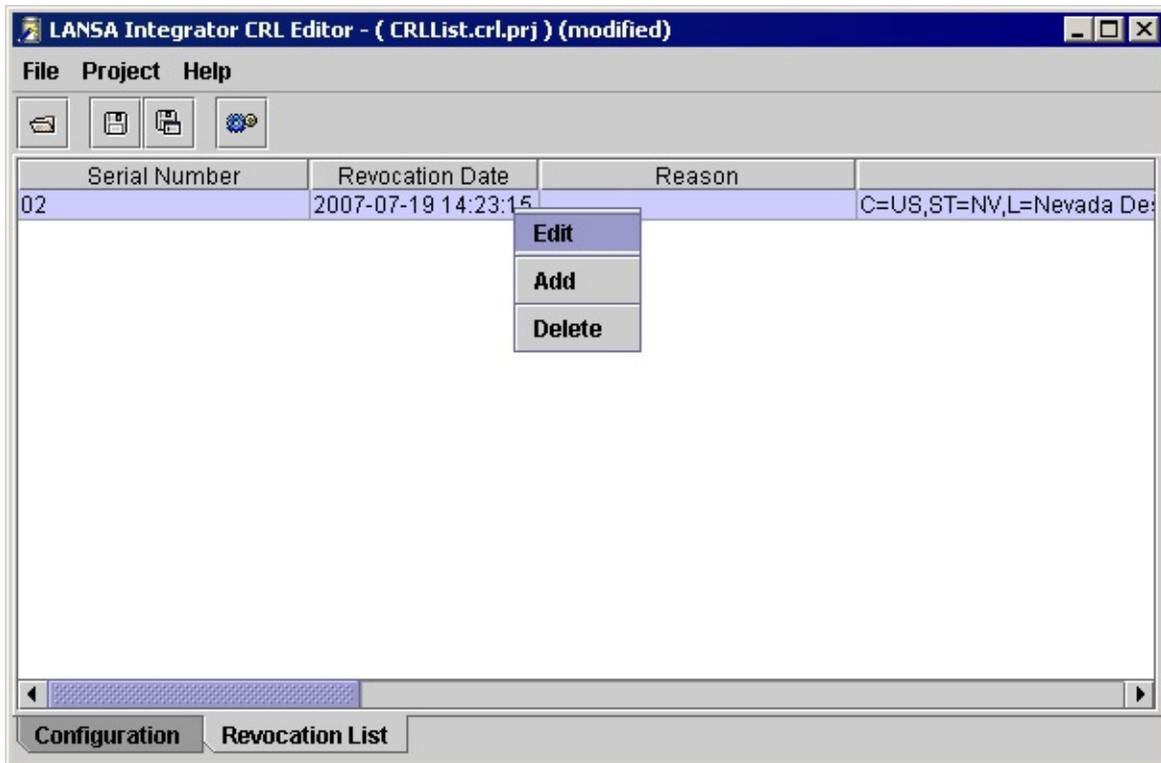
6. Select the Revocation List tab to maintain the list of certificates.



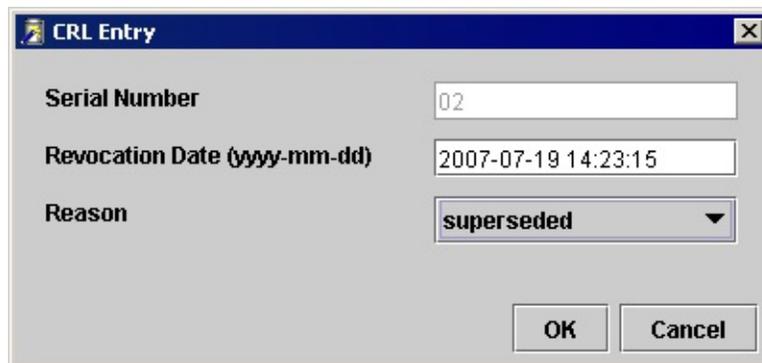
7. Use the Add pop-up menu item to add a certificate to the list.



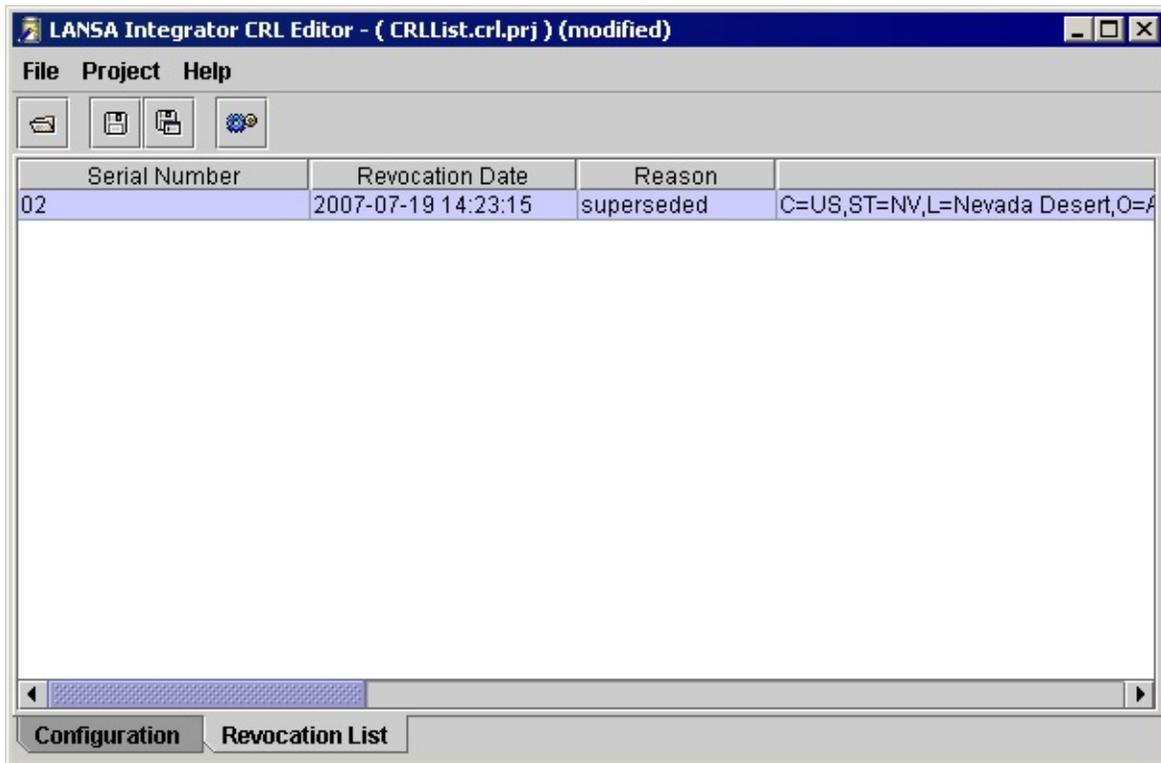
8. Use the Edit, Add and Delete pop-up menu items to maintain existing entries.



9. Use the Edit pop-up menu item to change the Revocation Date and Reason for revocation.

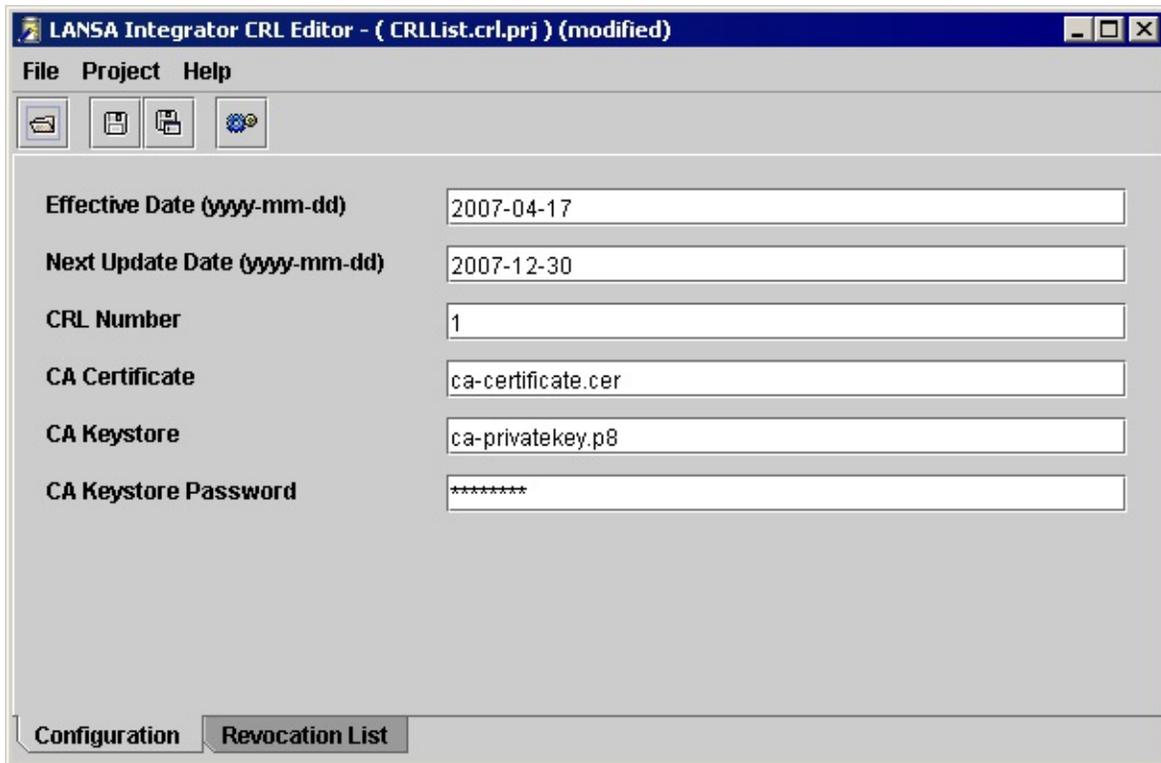


The Serial number, Revocation Date and Reason are the only values stored in the CRL file.



The Name column (fourth column on the right) is only used to identify the entry and is not included in the CRL file.

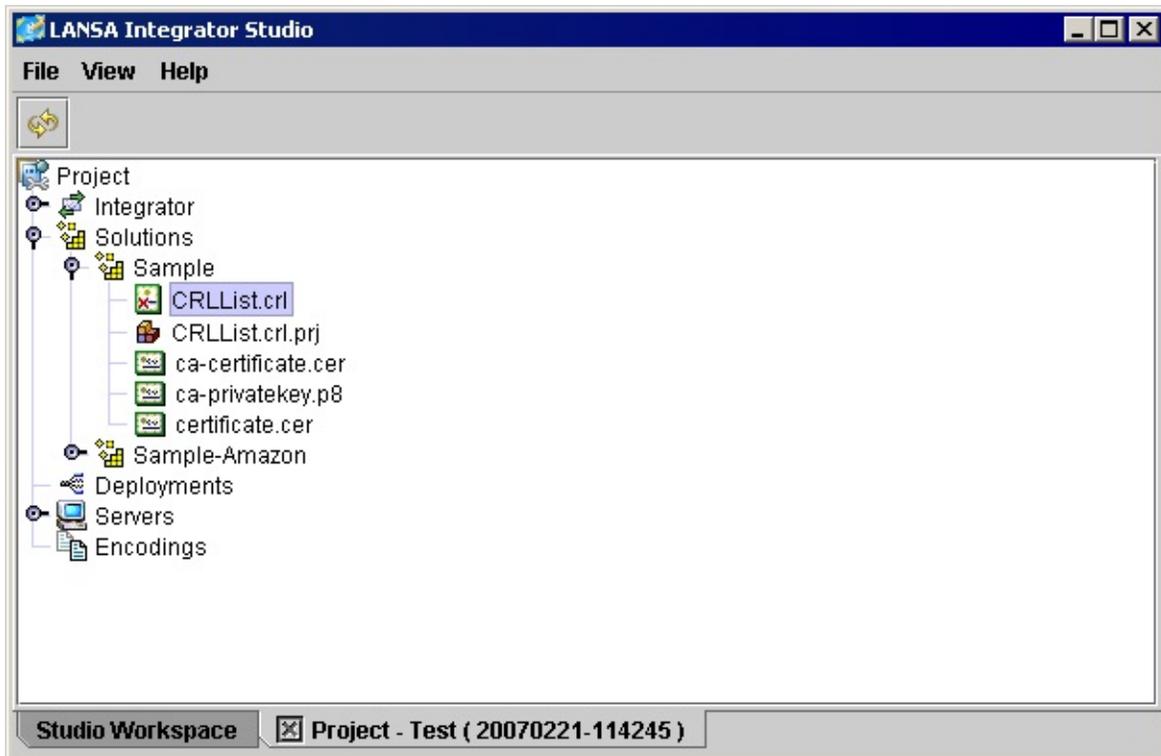
10. Press the Build button to create the CRL file.



A confirmation message is displayed:



11. Double click or open with associated application the CRL File.



The CRL file can be viewed using Windows CRL application.

Certificate Revocation List [?] [X]

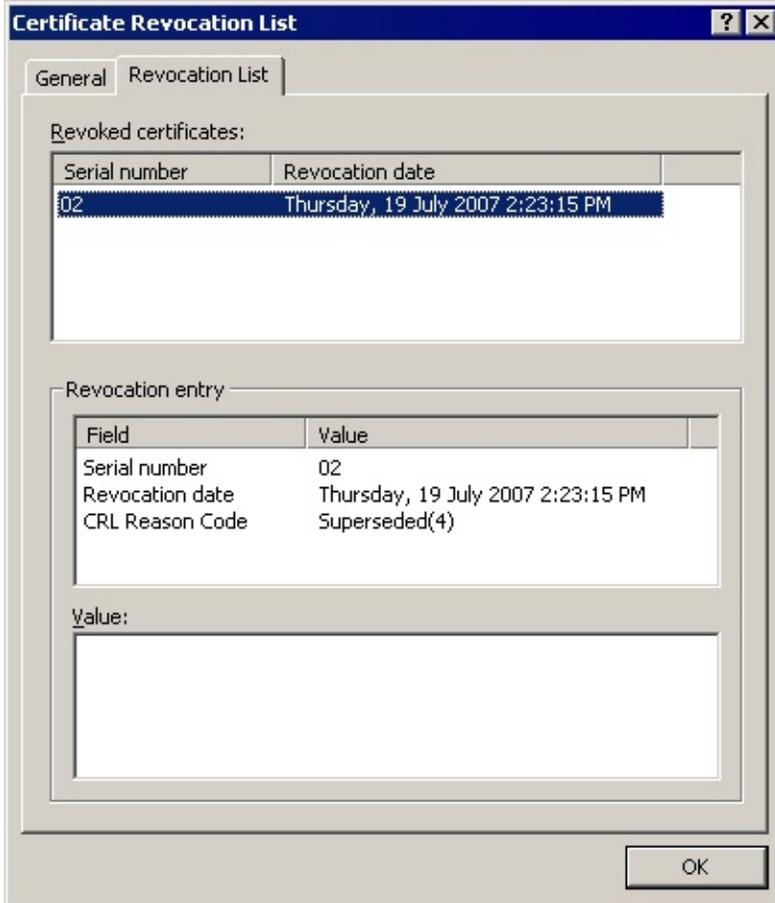
General | Revocation List

 **Certificate Revocation List Information**

Field	Value
Version	V2
Issuer	Road Runner, Rocket Powered Sy...
Effective date	Thursday, 19 July 2007 12:00:00 AM
Next update	Tuesday, 23 October 2007 12:00:...
Signature algorithm	1.2.840.113549.1.1.11
Authority Key Ident...	KeyID=35 f4 84 f7 8c fe 4e 1e 2f ...
CRL Number	1

Value:

OK



7.19.2 Distribute a CRL File

The CRL file can be distributed using various means.

- It could be emailed as an attachment to a known list of users.
- It could be downloaded from a HTTP server using the MIME type `application/pkix-crl`.

A CRL distribution URL can be included in each certificate created by the PKI Editor using the following PKI Editor project property.

`crl.distribution=http://www.mycompany.com/CRLList.crl`

`crl.distribution=http://www.mycompany.com/crllist.html`

7.20 SOAP Wizard

The SOAP Wizard is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The SOAP Wizard is an application that allows you to create and consume SOAP based Web Services.

The SOAP Agent Wizard is used to consume a WSDL document and to communicate with a remote server.

The SOAP Server Wizard is used to create a WSDL document and to publish a Web Service.

The SOAP Wizard requires a Java compiler, such as Oracle's Java SDK, to be installed on the PC.
Refer to [7.1.2 Set Java Compiler and Java Runtime](#).

Review the following topics:

- [7.20.1 Create a SOAP Agent Project](#)
- [7.20.2 Updating a SOAP Agent Project](#)
- [7.20.3 Create a SOAP Server Type](#)
- [7.20.4 Create a SOAP Server Project](#)
- [7.20.5 SOAP Server Wizard Naming Conventions](#)



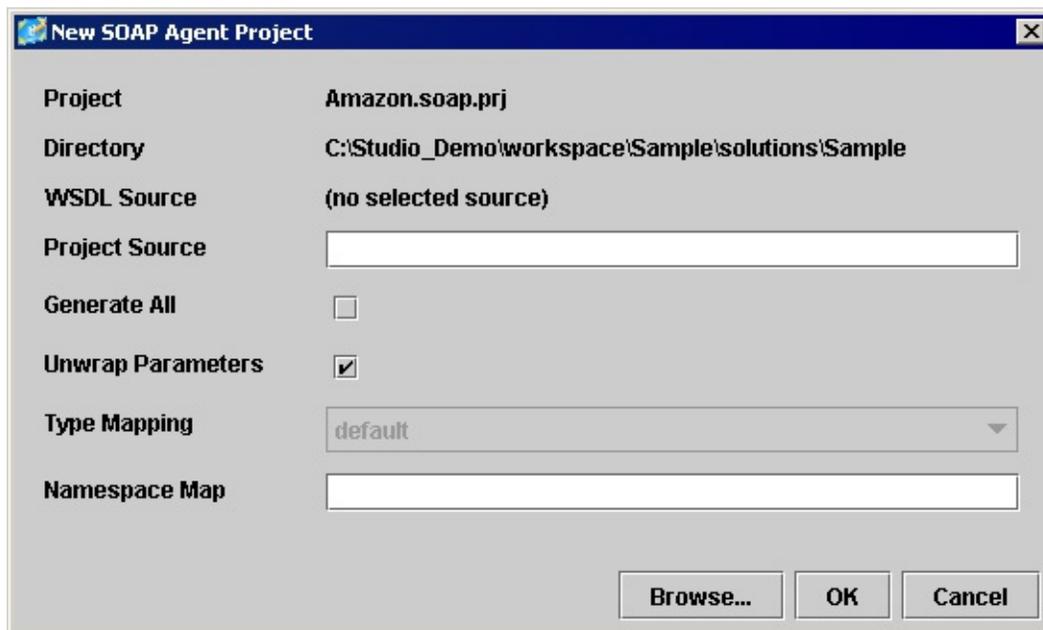
A step by step guide to using the SOAP Wizard is provided in the Integrator Tutorials commencing with [INT09A - SOAP Service - Define Server](#).

7.20.1 Create a SOAP Agent Project

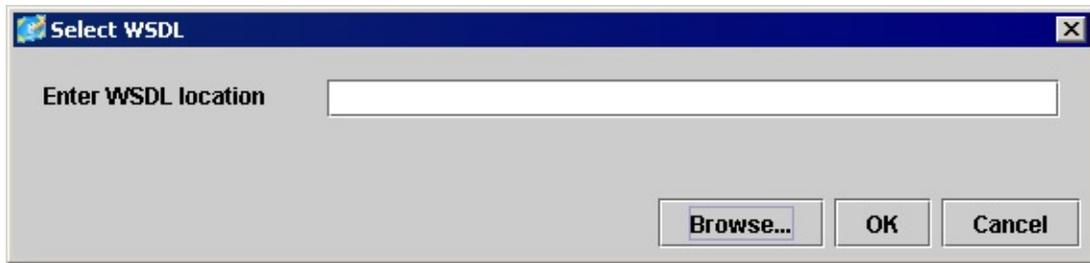
1. From the Project Solutions node select the *New Solution* menu item.
2. Select the *SOAP Agent Wizard* tool and enter or select the *Group* folder to receive the project file and WSDL document.
3. Enter the project File name. The extension *'.soap.prj'* is added automatically.



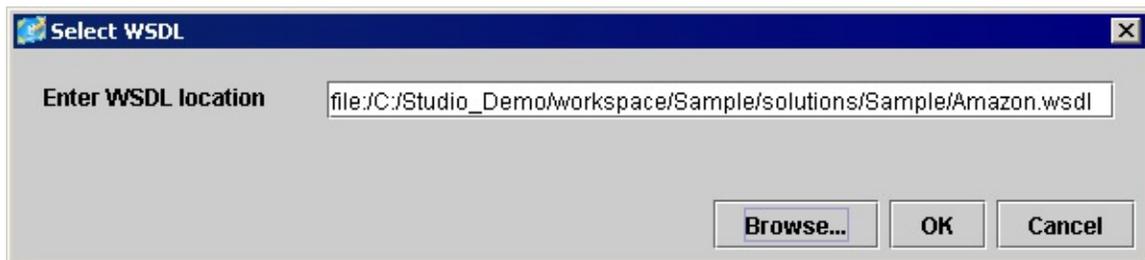
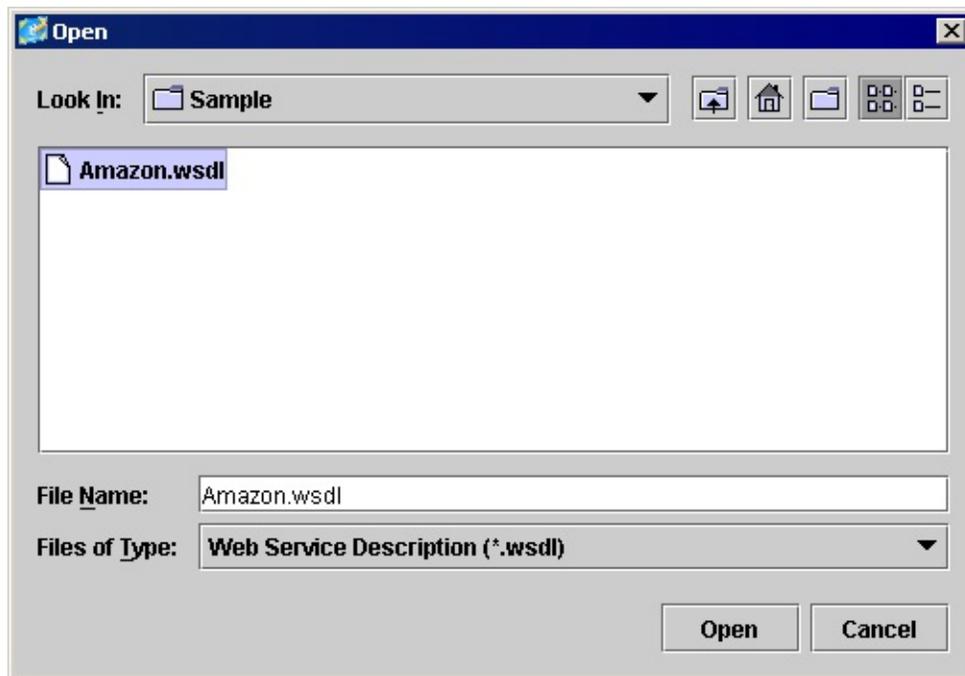
4. Press the OK button to select the WSDL file to be used.



5. Press the browse button to enter the URL to the WSDL file. If the WSDL file resides on the local file system press the next browse button to access the file system dialog box.



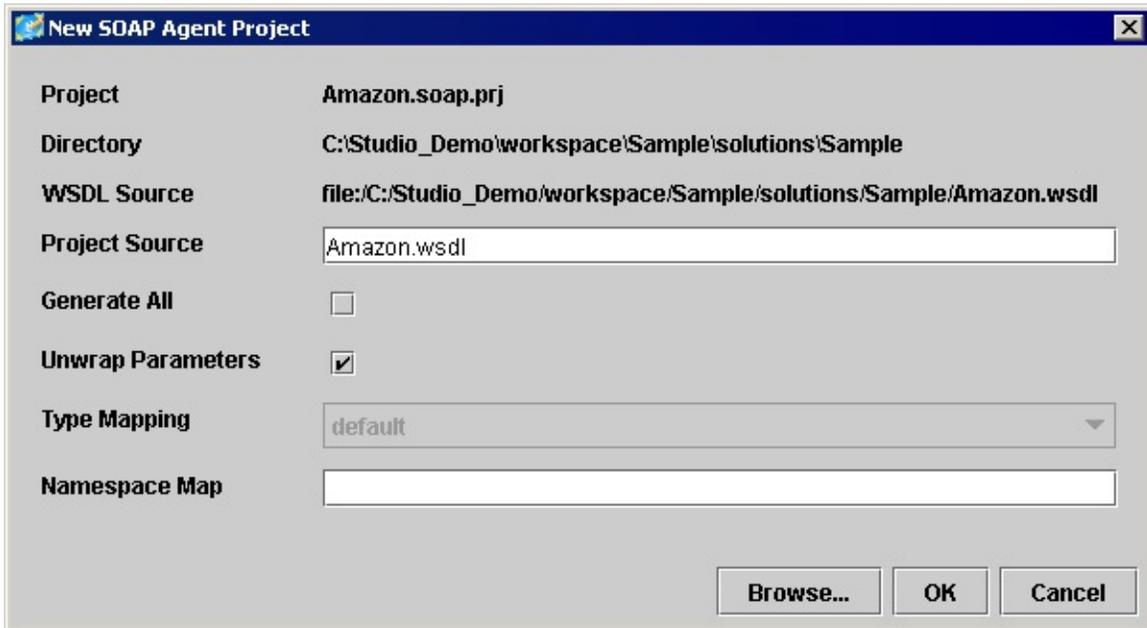
If the required URL is on the clipboard, then use CNTRL-V to paste it into the text field.



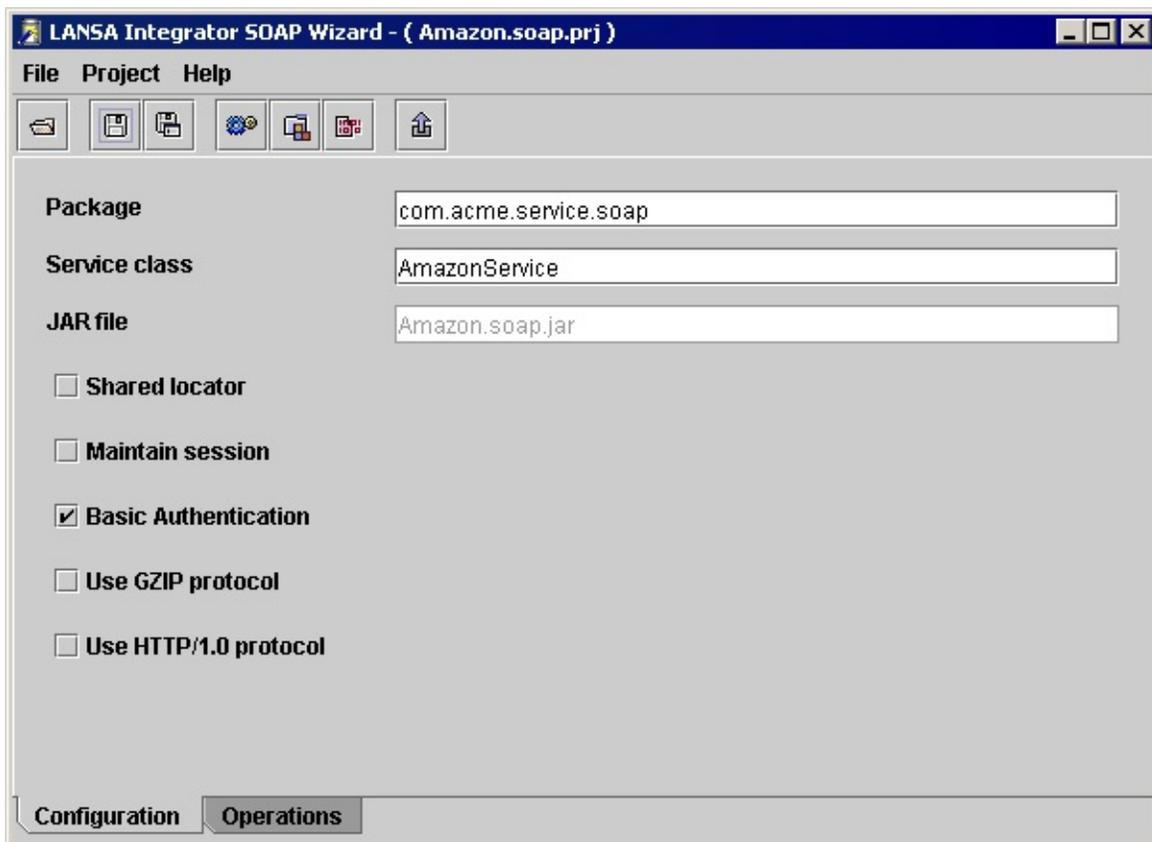
6. Press the OK button to return to the main dialog box. The Project Source field contains the file name that will be used to store the WSDL when it is retrieved using the URL. This file name can be changed.

The Wrapped Support check box is used to enable/disable "wrapped" document/literal style support.

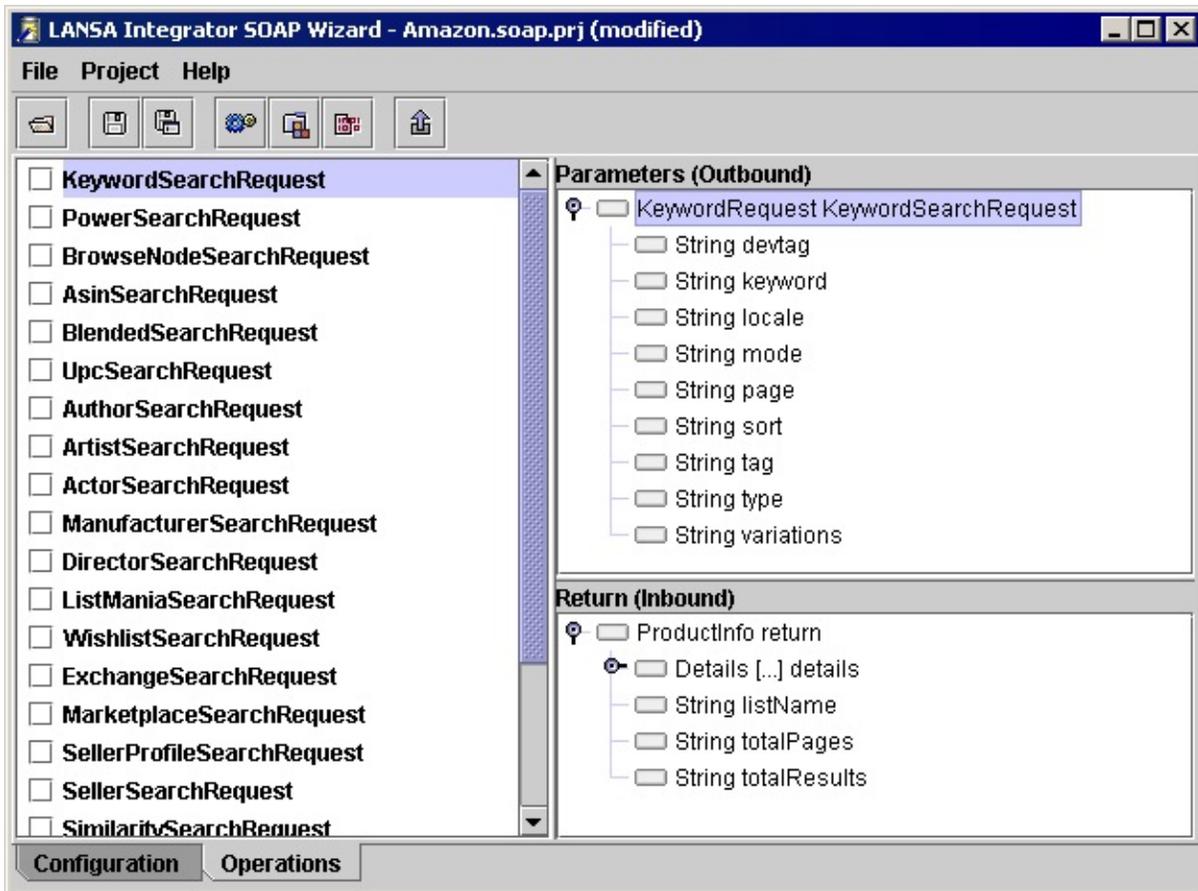
The Namespace Map is used to map all namespaces in the WSDL document to the same Java package name.



7. Press the OK button to create the project.



- The project name is used in the creation of the JAR file name and the service class name.
- The JAR file contains the Axis generated classes that handle the SOAP transaction and the generated service class.
- SOAP operations that are available are displayed in the left scroll panel.
- When an operation is selected the parameters and return parameter are displayed in right scroll panels.



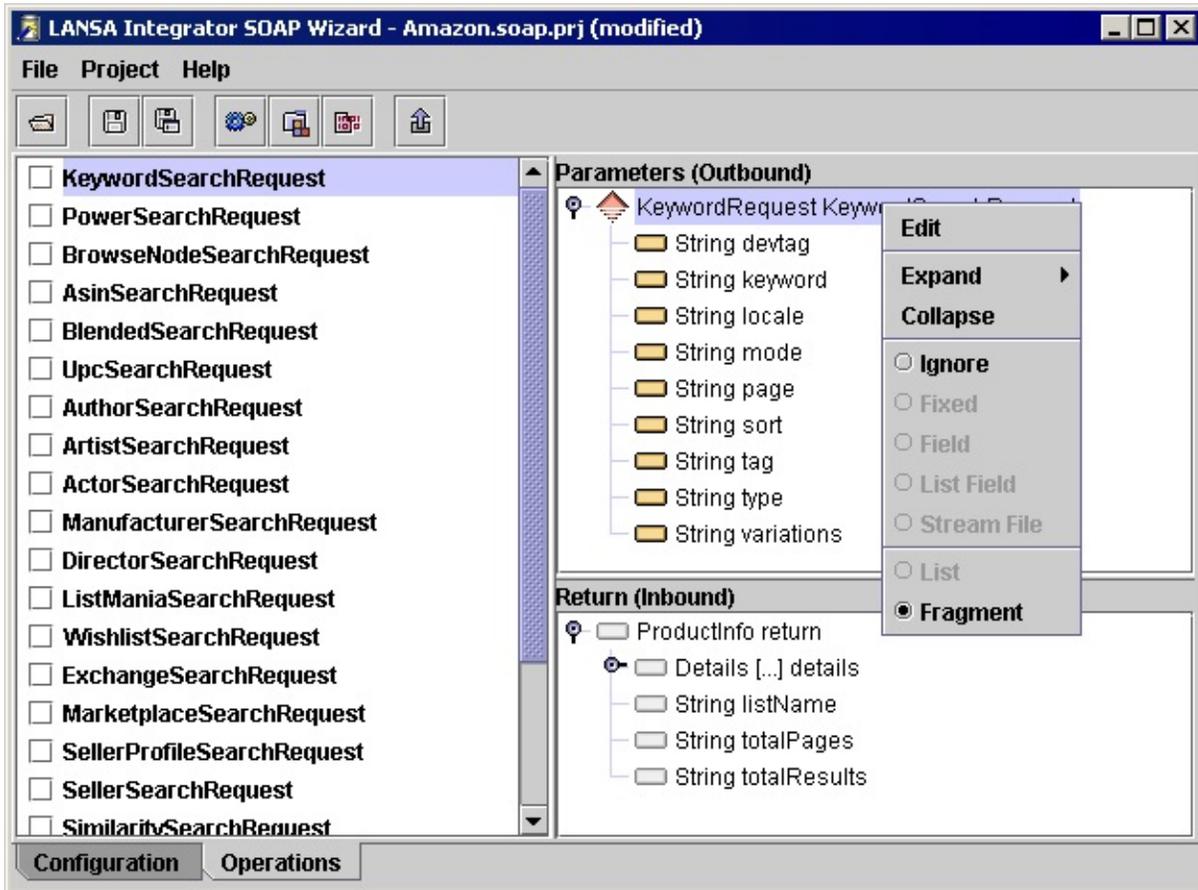
The parameter nodes can be marked to accept and handle data in different ways.

Different icons indicate the node types:

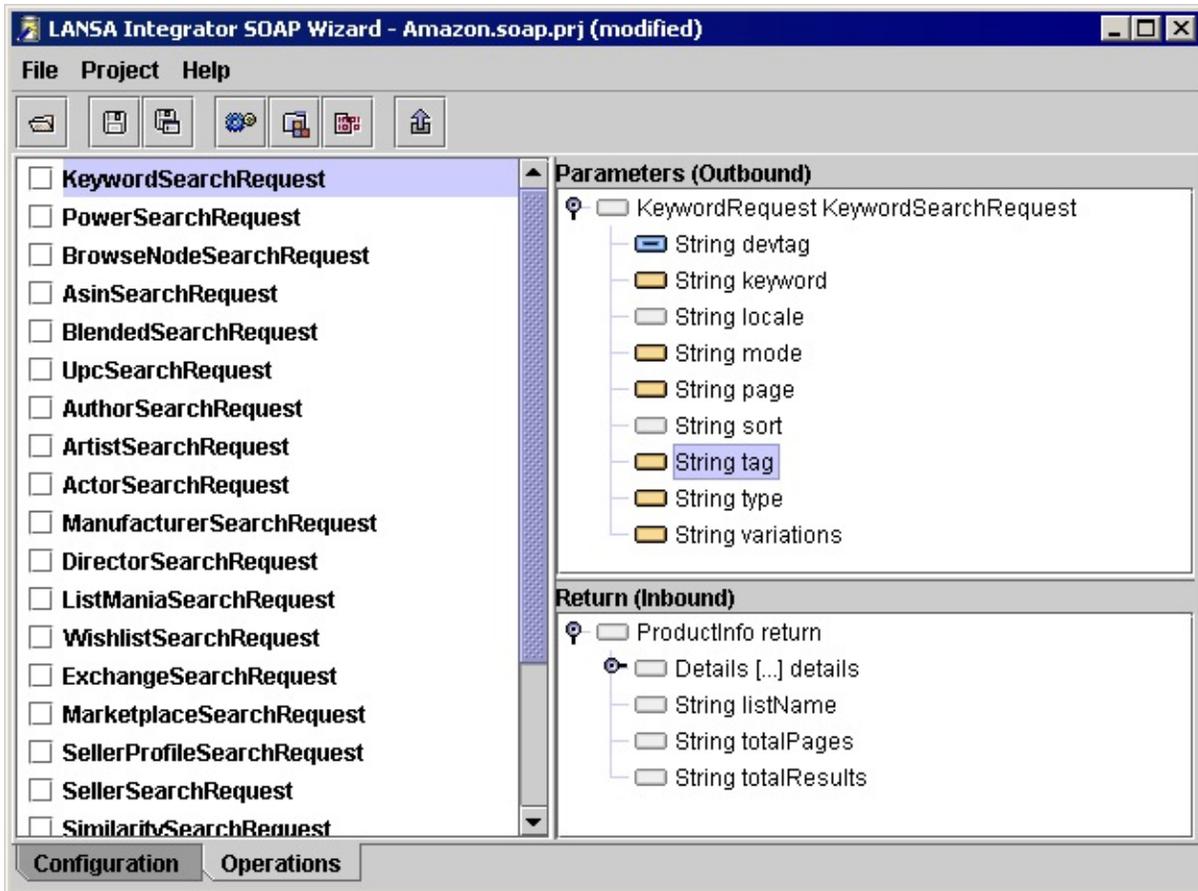
- Node is ignored.
- Node is a field.
- Node is a fixed value.
- Node is a list.

-  Node is a list field.
-  Node is a fragment.
-  Node is stream file.

In the following diagram the parameter has been marked as a fragment and a SET FRAGMENT command will be required to set the field values.

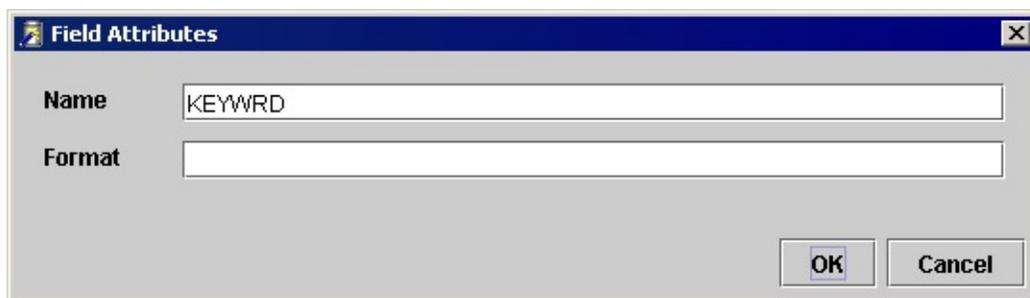


Also it is possible to leave the parameter unmarked and mark the fields. These fields will receive their values when the SET PARAMETER command is executed. The ignore fields will not have their values set and will default to null for Strings and objects and zero for primitive numbers.

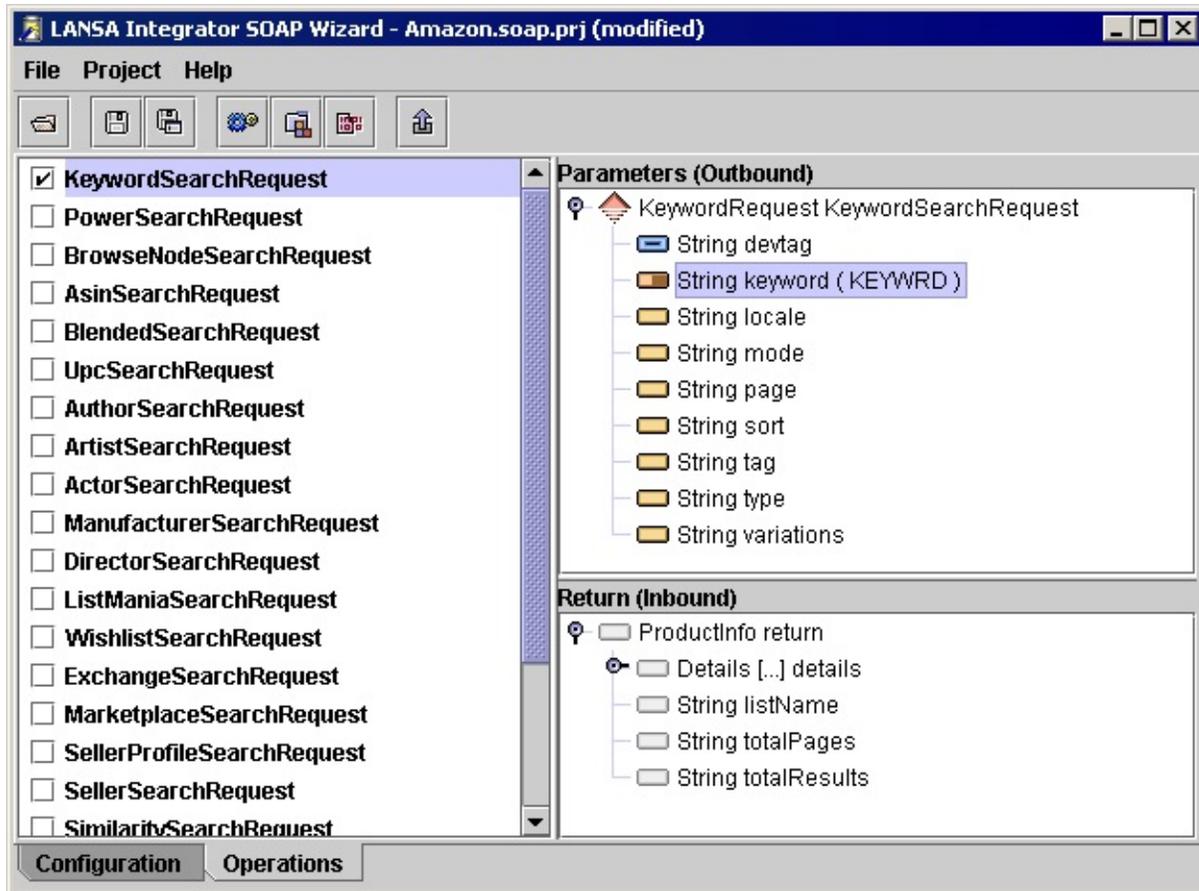


Fragment, list, list field and field nodes require the program field name for binding to occur between the program field and SOAP transaction. You can also use the [7.24 Repository Viewer](#) to drag and drop a LANSA field name onto a node.

8. Use the pop-up Edit menu or <Enter> key to access the entity attribute dialog.



9. Enter the program field name, fragment or list name and press the OK button.

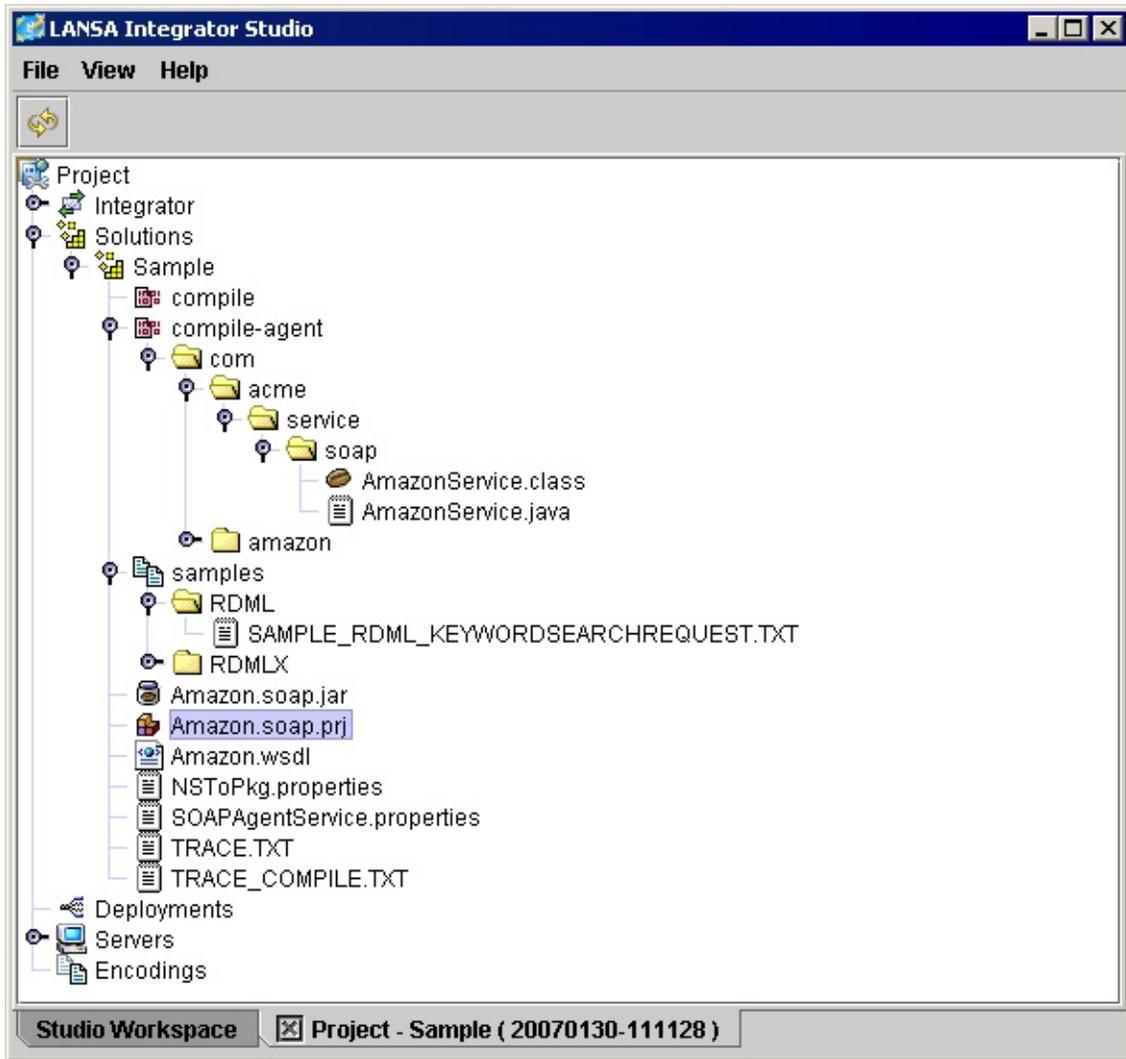


10. Press the build button to generate and compile the service, this service class will be added to the jar file.



All generated code and compiled classes exist in the compile-agent sub directory.

Files containing service properties are created in the solution directory and samples are created in the samples sub directory.

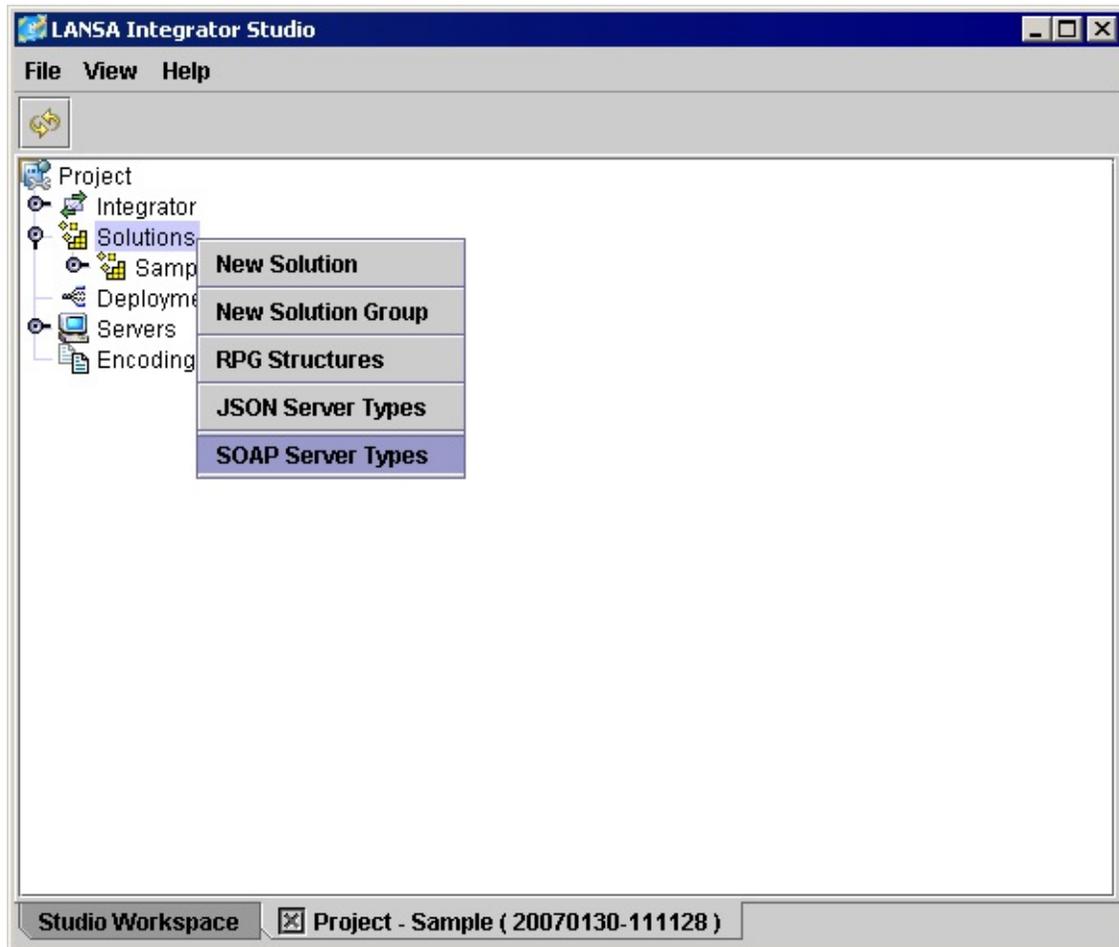


7.20.2 Updating a SOAP Agent Project

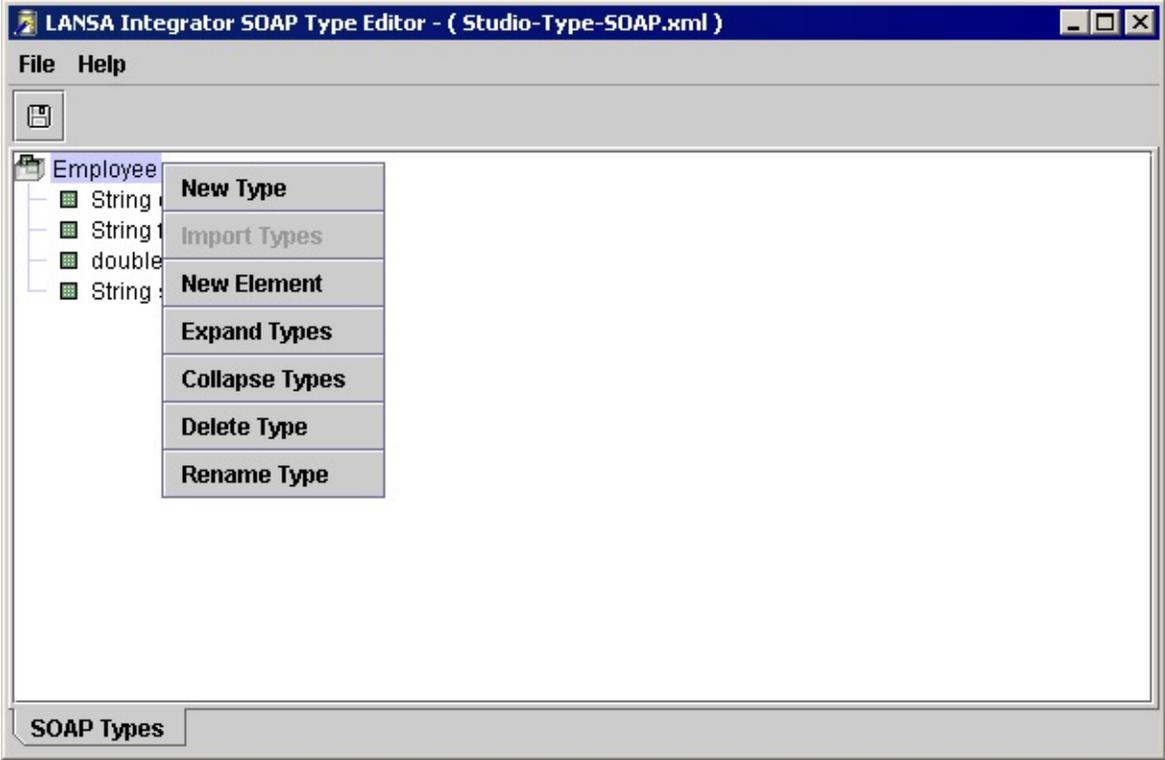
If the WSDL has changed and a new SOAP agent binding needs to be created, it is possible to update the existing project. Replace the existing WSDL file with the new WSDL file and use the 'Creating a SOAP Agent Project' steps again using the same project file name. The SOAP Agent Wizard will prompt with a message dialog asking to use the existing solution project file. Selecting 'Yes' will generate new SOAP agent files and preserve existing mappings.

7.20.3 Create a SOAP Server Type

From the Solution folder pop-up menu, select the *SOAP Server Types* menu item.

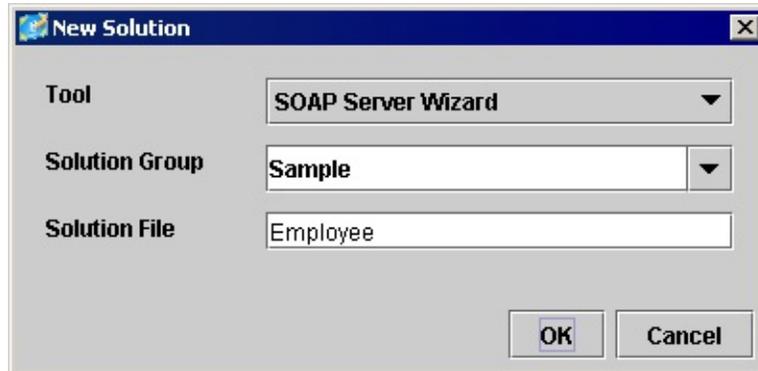


The SOAP Type Editor allows types to be created and maintained.

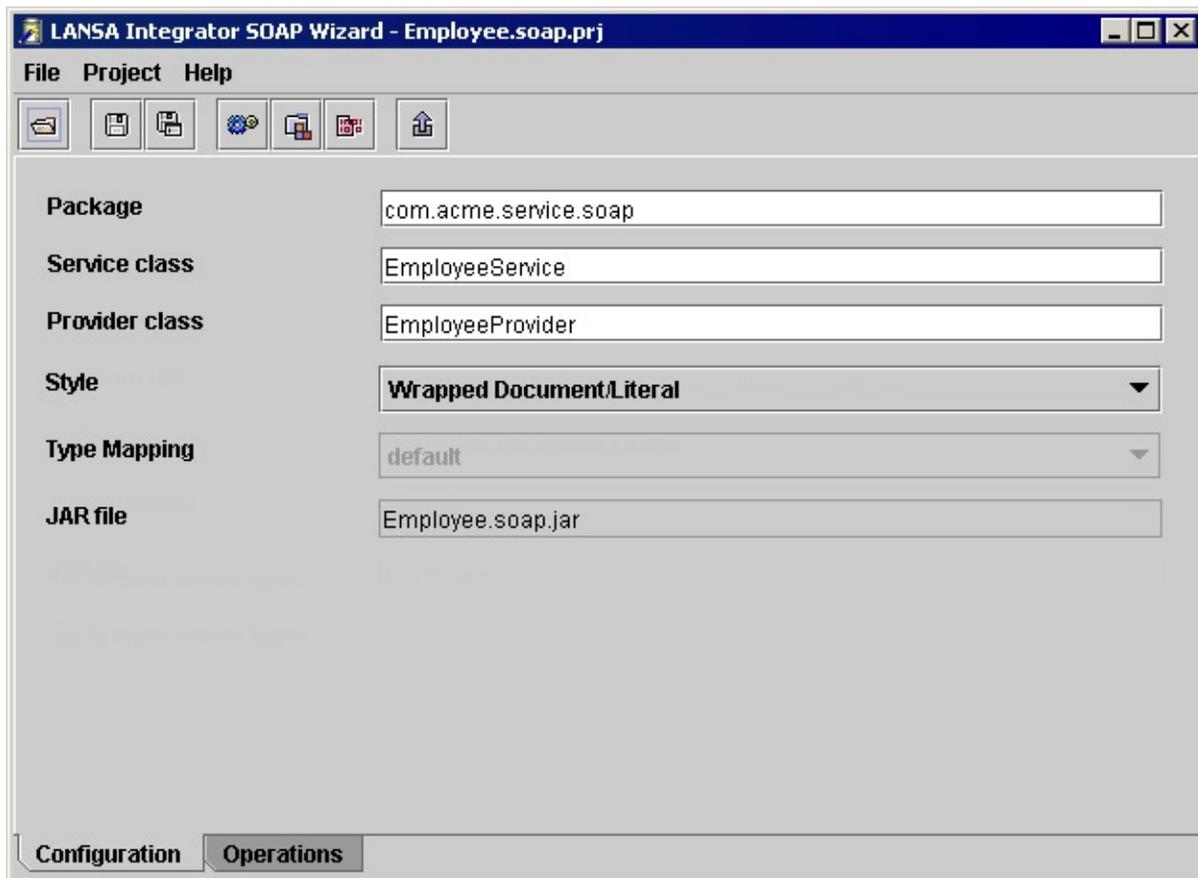


7.20.4 Create a SOAP Server Project

1. From the Project Solutions node select the *New Solution* menu item.
2. Select the *SOAP Server Wizard* tool and enter or select the *Group* folder to receive the project file.
3. Enter the project File name. The extension *'.soap.prj'* is added automatically.



4. Press the OK button.

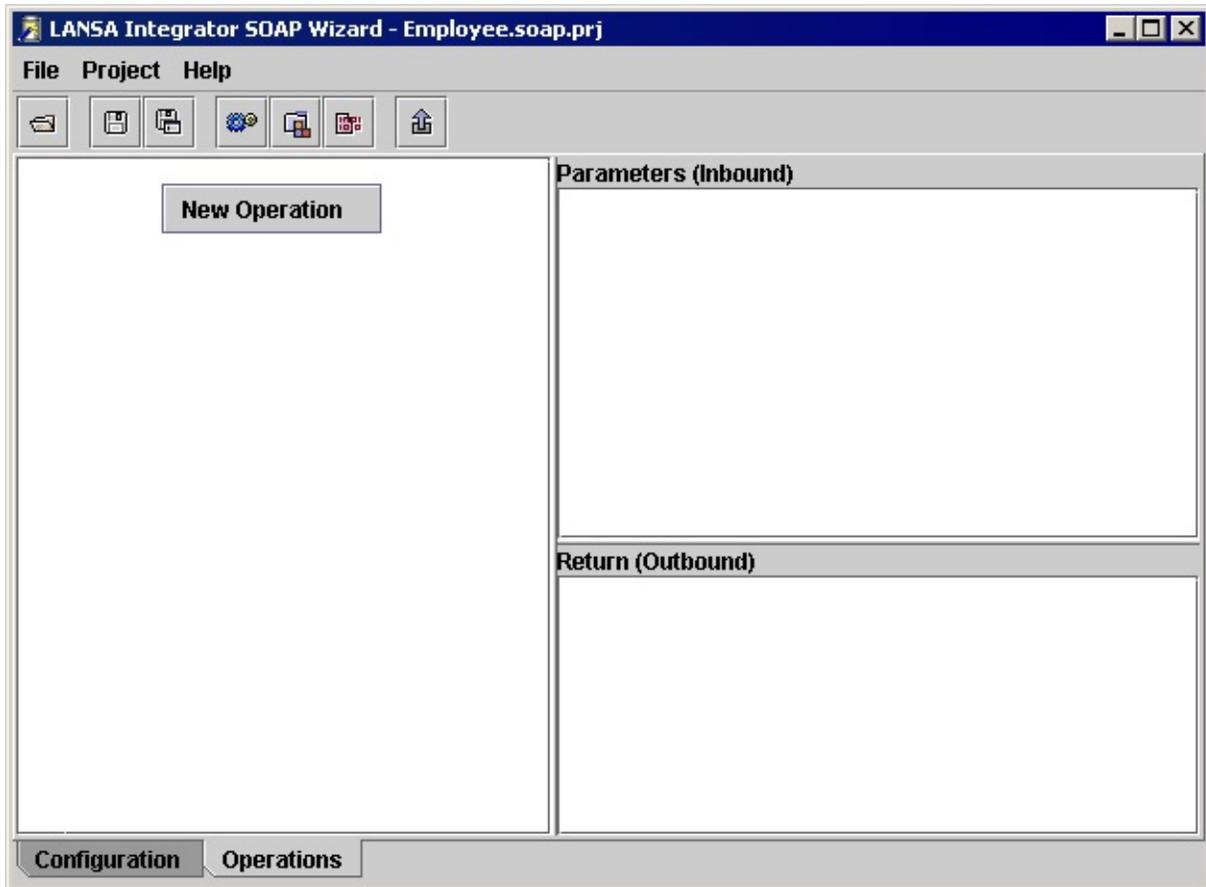


The project name is used in the creation of the JAR file name and the service and provider class names.

The JAR file contains the generated service and provider classes.

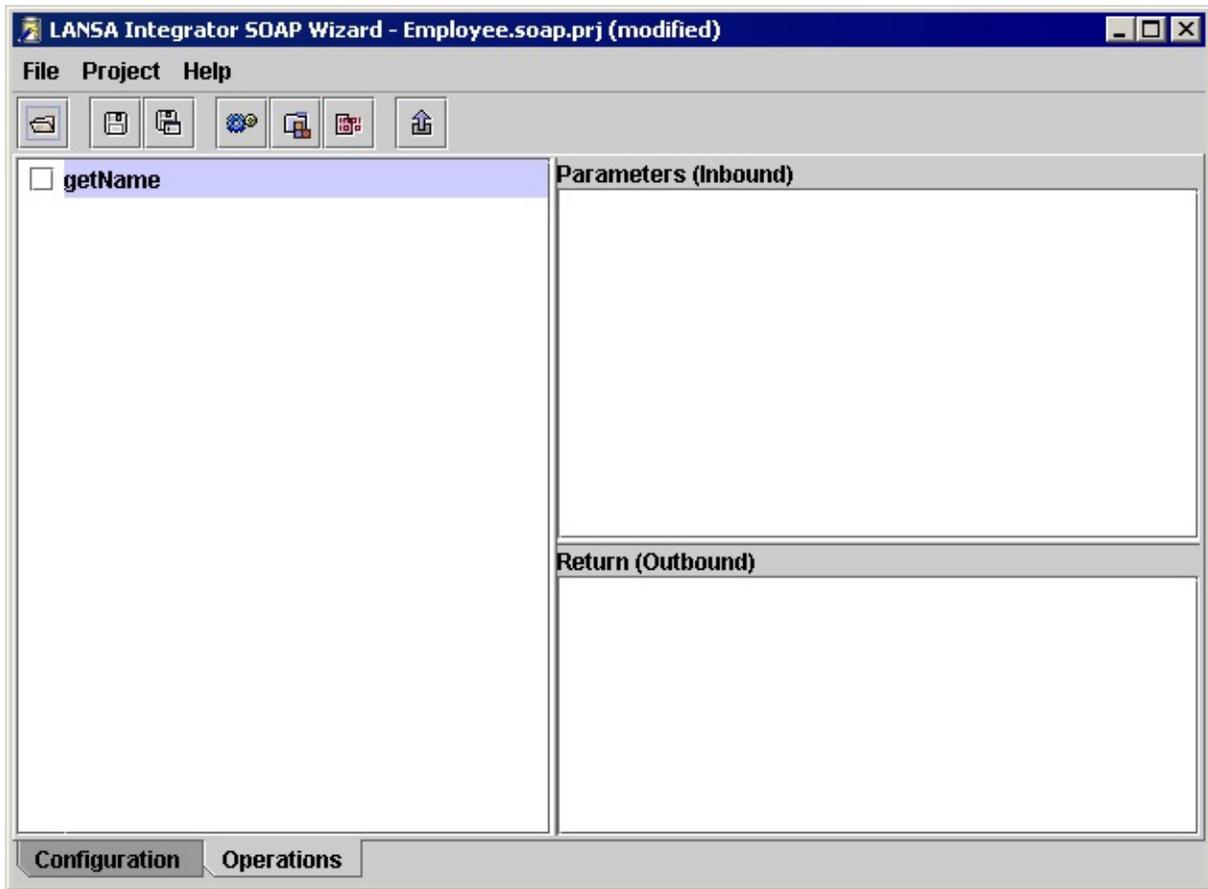
The end point URL is used in the generated WSDL file.

5. To add Operations to the server service, select the Operations tab and right click the operation panel.

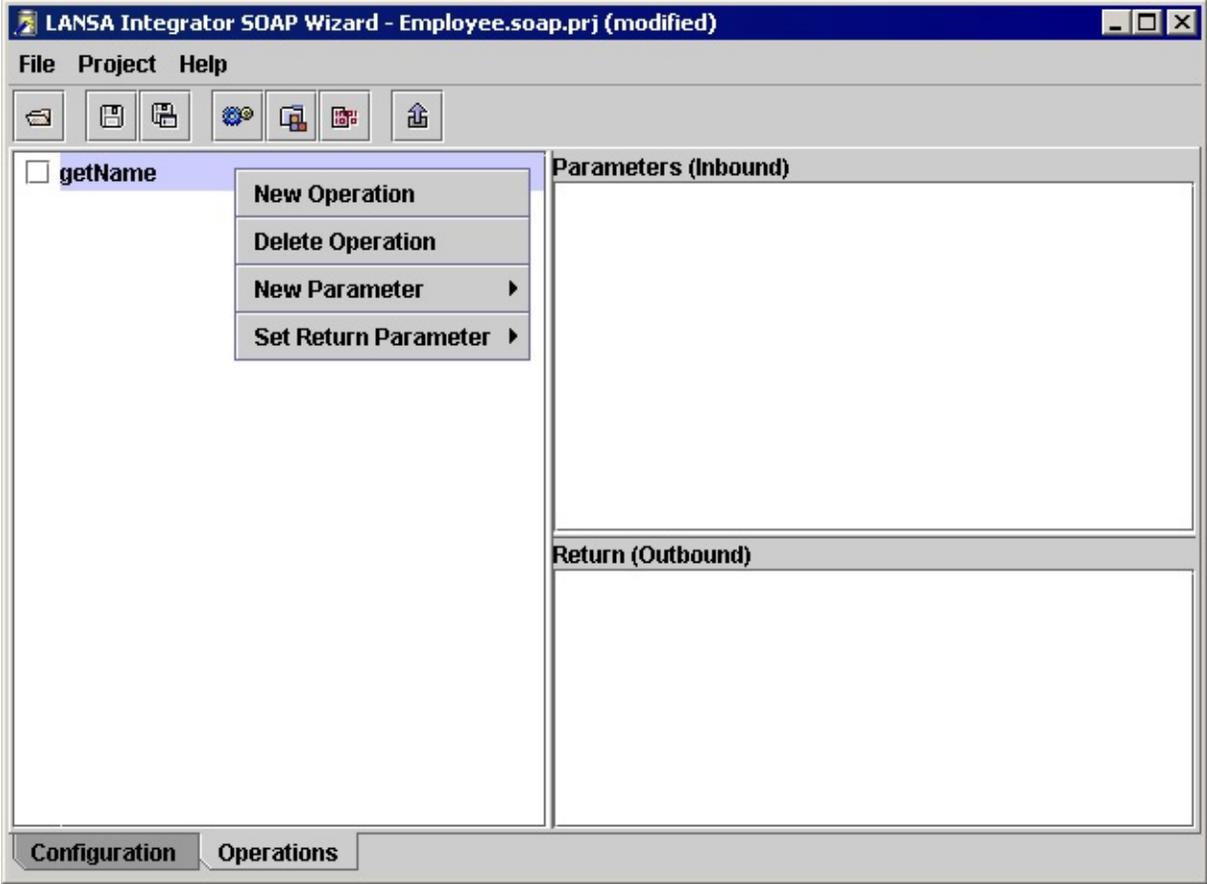


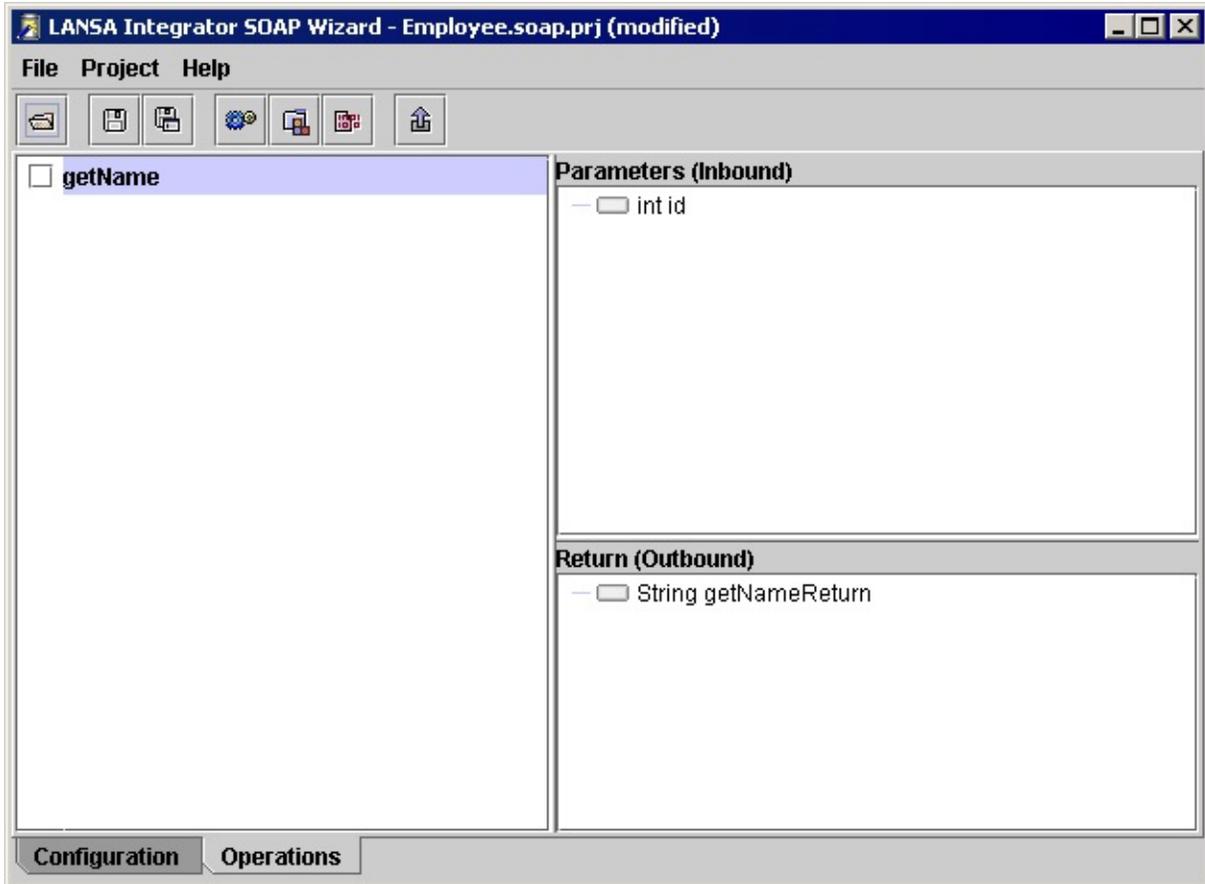
6. Enter Operation name.





7. To add operation parameters and to set the return parameter, right click the operation.



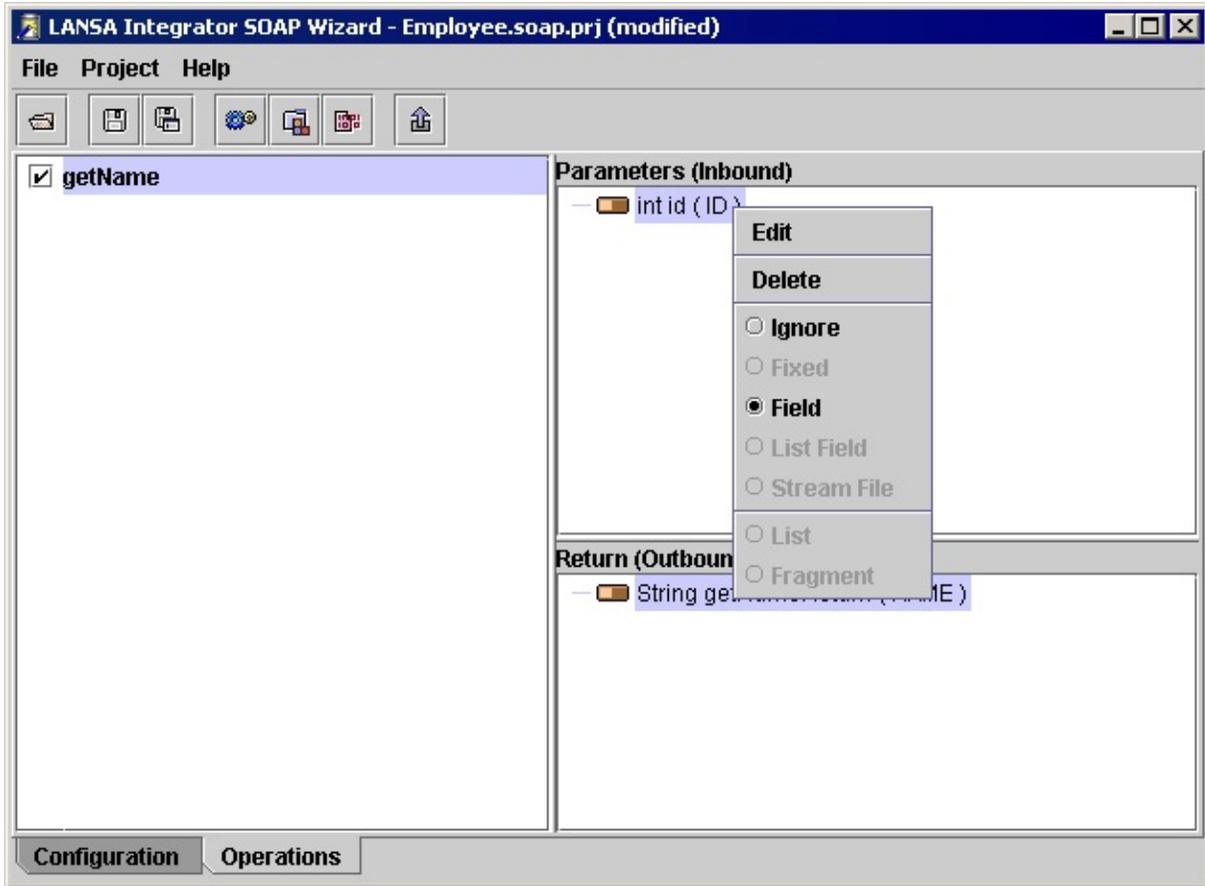


The parameter nodes can be marked to accept and handle data in different ways.

Different icons indicate the node types:

- Node is ignored.
- Node is a field.
- Node is a fixed value.
- Node is a list.
- Node is a list field.
- Node is a fragment.
- Node is stream file.

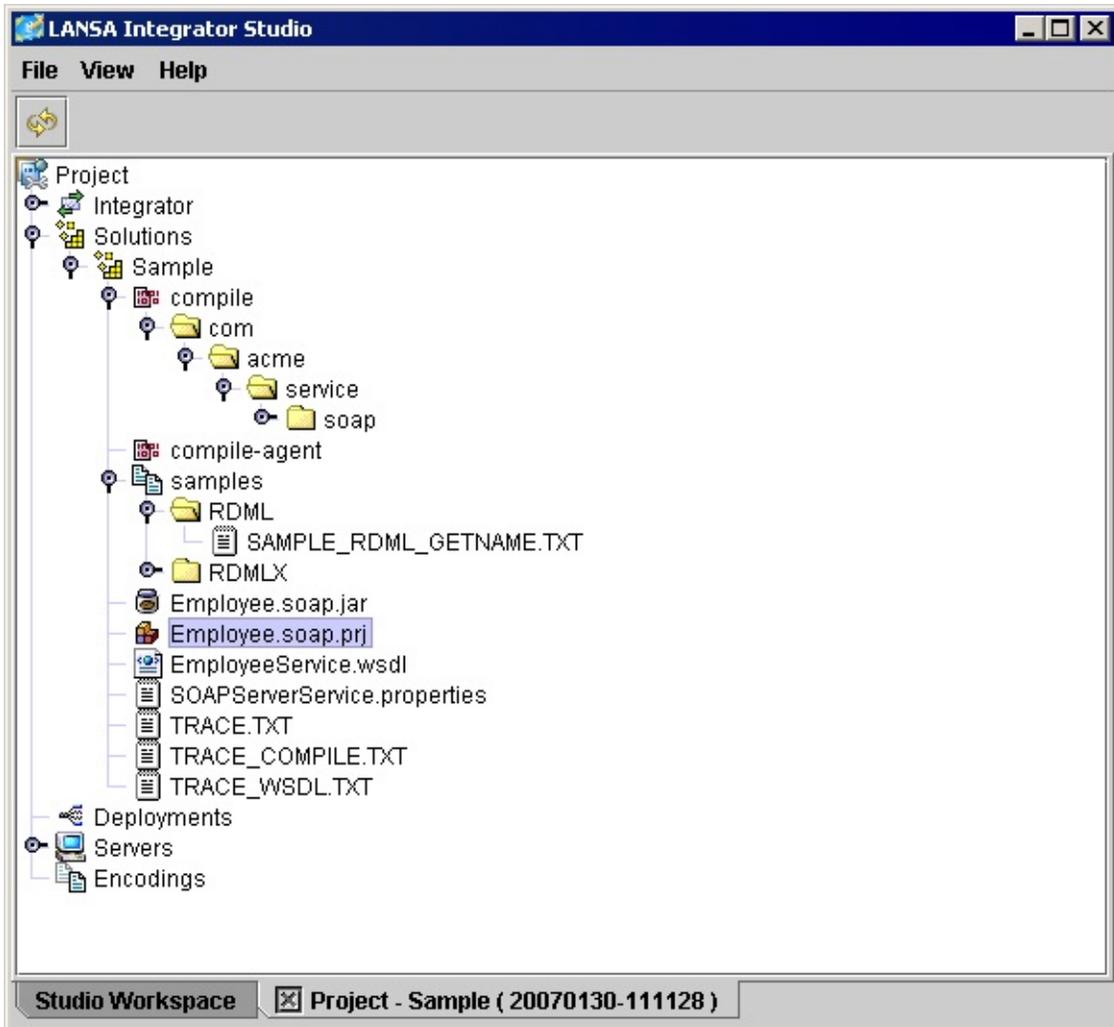
In the following diagram the parameter and return parameter have been marked as fields. You can also use the [7.24 Repository Viewer](#) to drag and drop a LANSA field name onto a node.



8. Press the build button to generate and compile the service. The service and provider classes will be added to the jar file.



All generated code and compiled classes exist in the compile sub directory. Files containing service properties and the WSDL document are created in the solution directory and samples are created in the samples sub directory.



7.20.5 SOAP Server Wizard Naming Conventions

Service Operation Name conventions:

- Start operation name with a lowercase letter as in the following example:

```
getEmployees  
selectActiveAccount
```

SOAP Server Type conventions.

- Start type with an uppercase letter.
- Start element/variable name with at least two lowercase letters.
- Do not start element/variable name with a number.
- Do not embed numbers in element/variable name. If a number is embedded then uppercase the first letter after the number.
- Do not use an element/variable name of typeDesc. The Apache Axis WSDL2Java program generates bean classes with a member variable with this name.

Example of element/variable names:

```
date  
amount  
sentDate  
active_status
```

- An Apache Axis generated bean class can only have a maximum of 254 member variables. This is due to the Java constructor limit of 254 parameters.
- Axis creates JSR-101 compliant Java names from SOAP XML names using the following method:

```
org.apache.axis.utils.JavaUtils.xmlNameToJava ( xmlName )
```

Also see

[Example of a SOAP Server Wizard generated bean class](#)

Example of Apache Axis WSDL2Java generated bean class

Example of a SOAP Server Wizard generated bean class

```
/**
 * Order.java
 *
 * This file was generated by LANSA Integrator SOAP Server Wizard
 */
package com.acme.service.soap ;

public class Order implements java.io.Serializable
{
    private String date ;
    private String dateOfOrder ;
    private int order_id ;
    private OrderLine[] orders ;
    private long total ;

    /**
     * Sets the date value for this Order.
     *
     * @param date
     */
    public void setDate ( String date )
    {
        this.date = date ;
    }

    /**
     * Gets the date value for this Order.
     *
     * @return date
     */
    public String getDate ()
    {
        return date ;
    }
}
```

```
/**
 * Sets the dateOfOrder value for this Order.
 *
 * @param dateOfOrder
 */
public void setDateOfOrder ( String dateOfOrder )
{
    this.dateOfOrder = dateOfOrder ;
}
```

```
/**
 * Gets the dateOfOrder value for this Order.
 *
 * @return dateOfOrder
 */
public String getDateOfOrder ()
{
    return dateOfOrder ;
}
```

```
/**
 * Sets the order_id value for this Order.
 *
 * @param order_id
 */
public void setOrder_id ( int order_id )
{
    this.order_id = order_id ;
}
```

```
/**
 * Gets the order_id value for this Order.
 *
 * @return order_id
 */
public int getOrder_id ()
{
    return order_id ;
}
```

```
/**
 * Sets the orders value for this Order.
 *
 * @param orders
 */
public void setOrders ( OrderLine[] orders )
{
    this.orders = orders ;
}
```

```
/**
 * Gets the orders value for this Order.
 *
 * @return orders
 */
public OrderLine[] getOrders ()
{
    return orders ;
}
```

```
/**
 * Sets the total value for this Order.
 *
 * @param total
 */
public void setTotal ( long total )
{
    this.total = total ;
}
```

```
/**
 * Gets the total value for this Order.
 *
 * @return total
 */
public long getTotal ()
{
    return total ;
}
```

}
}

Example of Apache Axis WSDL2Java generated bean class

```
/**
 * Order.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis 1.3 Oct 05, 2005 (05:23:37 EDT) WSDL2Java emitter.
 */

package com.acme.service.soap;

public class Order implements java.io.Serializable
{
    private java.lang.String date;
    private java.lang.String dateOfOrder;
    private int order_id;
    private com.acme.service.soap.OrderLine[] orders;
    private long total;

    public Order ()
    {
    }

    public Order (
        java.lang.String date,
        java.lang.String dateOfOrder,
        int order_id,
        com.acme.service.soap.OrderLine[] orders,
        long total)
    {
        this.date = date;
        this.dateOfOrder = dateOfOrder;
        this.order_id = order_id;
        this.orders = orders;
        this.total = total;
    }
}
```

```
/**
 * Gets the date value for this Order.
 *
 * @return date
 */
public java.lang.String getDate()
{
    return date;
}

/**
 * Sets the date value for this Order.
 *
 * @param date
 */
public void setDate(java.lang.String date)
{
    this.date = date;
}

/**
 * Gets the dateOfOrder value for this Order.
 *
 * @return dateOfOrder
 */
public java.lang.String getDateOfOrder()
{
    return dateOfOrder;
}

/**
 * Sets the dateOfOrder value for this Order.
 *
 * @param dateOfOrder
 */
public void setDateOfOrder(java.lang.String dateOfOrder)
{
    this.dateOfOrder = dateOfOrder;
}
```

```
/**
 * Gets the order_id value for this Order.
 *
 * @return order_id
 */
public int getOrder_id()
{
    return order_id;
}

/**
 * Sets the order_id value for this Order.
 *
 * @param order_id
 */
public void setOrder_id(int order_id)
{
    this.order_id = order_id;
}

/**
 * Gets the orders value for this Order.
 *
 * @return orders
 */
public com.acme.service.soap.OrderLine[] getOrders()
{
    return orders;
}

/**
 * Sets the orders value for this Order.
 *
 * @param orders
 */
public void setOrders(com.acme.service.soap.OrderLine[] orders)
{
    this.orders = orders;
}
```

```

}

/**
 * Gets the total value for this Order.
 *
 * @return total
 */
public long getTotal()
{
    return total;
}

/**
 * Sets the total value for this Order.
 *
 * @param total
 */
public void setTotal(long total)
{
    this.total = total;
}

private java.lang.Object __equalsCalc = null;

public synchronized boolean equals(java.lang.Object obj)
{
    if (!(obj instanceof Order)) return false;
    Order other = (Order) obj;
    if (obj == null) return false;
    if (this == obj) return true;
    if (__equalsCalc != null) {
        return (__equalsCalc == obj);
    }
    __equalsCalc = obj;
    boolean _equals;
    equals = true &&
        ((this.date==null && other.getDate()==null) ||
        (this.date!=null &&
        this.date.equals(other.getDate())) &&

```

```

        ((this.dateOfOrder==null && other.getDateOfOrder()==null) ||
        (this.dateOfOrder!=null &&
        this.dateOfOrder.equals(other.getDateOfOrder())) &&
        this.order_id == other.getOrder_id() &&
        ((this.orders==null && other.getOrders()==null) ||
        (this.orders!=null &&
        java.util.Arrays.equals(this.orders, other.getOrders())) &&
        this.total == other.getTotal());
    __equalsCalc = null;

    return _equals;
}

private boolean __hashCodeCalc = false;

public synchronized int hashCode()
{
    if (__hashCodeCalc)
    {
        return 0;
    }

    __hashCodeCalc = true;
    int _hashCode = 1;

    if (getDate() != null)
    {
        _hashCode += getDate().hashCode();
    }

    if (getDateOfOrder() != null)
    {
        _hashCode += getDateOfOrder().hashCode();
    }

    _hashCode += getOrder_id();

    if (getOrders() != null)
    {

```

```

    for (int i=0; i<java.lang.reflect.Array.getLength(getOrders()); i++)
    {
        java.lang.Object obj = java.lang.reflect.Array.get(getOrders(), i);
        if (obj != null &&!obj.getClass().isArray())
        {
            _hashCode += obj.hashCode();
        }
    }
}

__hashCode += new Long(getTotal()).hashCode();
__hashCodeCalc = false;

return _hashCode;
}

// Type metadata
private static org.apache.axis.description.TypeDesc typeDesc =
    new org.apache.axis.description.TypeDesc(Order.class, true);

static
{
    typeDesc.setXmlType(new javax.xml.namespace.QName("http://soap.ser
org.apache.axis.description.ElementDesc elemField = new org.apache.axi
elemField.setFieldName("date");
elemField.setXmlName(new javax.xml.namespace.QName("", "date"));
elemField.setXmlType(new javax.xml.namespace.QName("http://www.w
elemField.setNillable(true);
typeDesc.addFieldDesc(elemField);
elemField = new org.apache.axis.description.ElementDesc();
elemField.setFieldName("dateOfOrder");
elemField.setXmlName(new javax.xml.namespace.QName("", "dateOfOr
elemField.setXmlType(new javax.xml.namespace.QName("http://www.w
elemField.setNillable(true);
typeDesc.addFieldDesc(elemField);
elemField = new org.apache.axis.description.ElementDesc();
elemField.setFieldName("order_id");
elemField.setXmlName(new javax.xml.namespace.QName("", "order_id"
elemField.setXmlType(new javax.xml.namespace.QName("http://www.w

```

```

elemField.setNillable(false);
typeDesc.addFieldDesc(elemField);
elemField = new org.apache.axis.description.ElementDesc();
elemField.setFieldName("orders");
elemField.setXmlName(new javax.xml.namespace.QName("", "orders"));
elemField.setXmlType(new javax.xml.namespace.QName("http://soap.sei
elemField.setNillable(true);
typeDesc.addFieldDesc(elemField);
elemField = new org.apache.axis.description.ElementDesc();
elemField.setFieldName("total");
elemField.setXmlName(new javax.xml.namespace.QName("", "total"));
elemField.setXmlType(new javax.xml.namespace.QName("http://www.w
elemField.setNillable(false);
typeDesc.addFieldDesc(elemField);
}

```

```

/**
 * Return type metadata object
 */
public static org.apache.axis.description.TypeDesc getTypeDesc()
{
    return typeDesc;
}

```

```

/**
 * Get Custom Serializer
 */
public static org.apache.axis.encoding.Serializer getSerializer (
    java.lang.String mechType,
    java.lang.Class _javaType,
    javax.xml.namespace.QName _xmlType )
{
    return new org.apache.axis.encoding.ser.BeanSerializer(
        _javaType, _xmlType, typeDesc);
}

```

```

/**
 * Get Custom Deserializer
 */

```

```
public static org.apache.axis.encoding.Deserializer getDeserializer (
    java.lang.String mechType,
    java.lang.Class _javaType,
    javax.xml.namespace.QName _xmlType )
{
    return new org.apache.axis.encoding.ser.BeanDeserializer(
        _javaType, _xmlType, typeDesc);
}
}
```

7.21 XML Binding Wizard

The XML Binding Wizard is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The XML Binding Wizard is an application that allows you to map XML elements and attributes to LANSAs fields and lists. The Wizard will generate code to be used by XML binding services to read and write XML documents.

The XML Binding Wizard requires a Java compiler, such as Oracle's Java SDK, to be installed on the PC.
Refer to [7.1.2 Set Java Compiler and Java Runtime](#).

Review the following topics:

- [7.21.1 Sample XML Document](#)
- [7.21.2 Changing Sample XML Document](#)
- [7.21.3 Create an XML Binding Project](#)

7.21.1 Sample XML Document

The XML Binding Wizard requires a sample XML document. It does not directly use an XML Schema or DTD file.

It is imperative that your sample XML document contains all XML elements and attributes required for mapping.

The sample XML document must be well formed and follow the rules and syntax of XML documents.

The sample XML file may be provided by a third party application or you may need to create your own XML document.

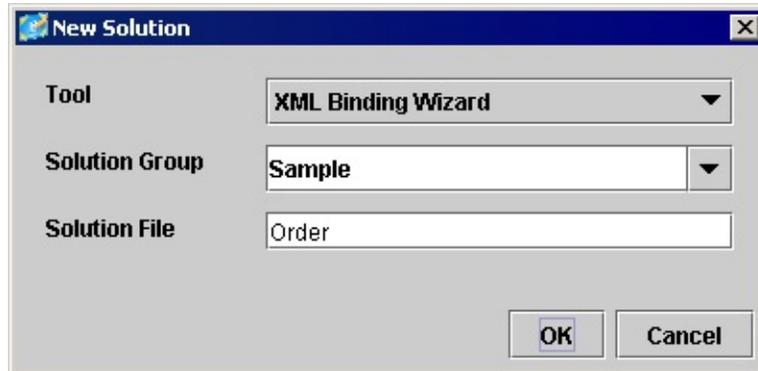
You can create a sample XML file using a text editor or use the [7.12 XMLSchema Viewer](#) to generate a sample XML file from an XML Schema file.

7.21.2 Changing Sample XML Document

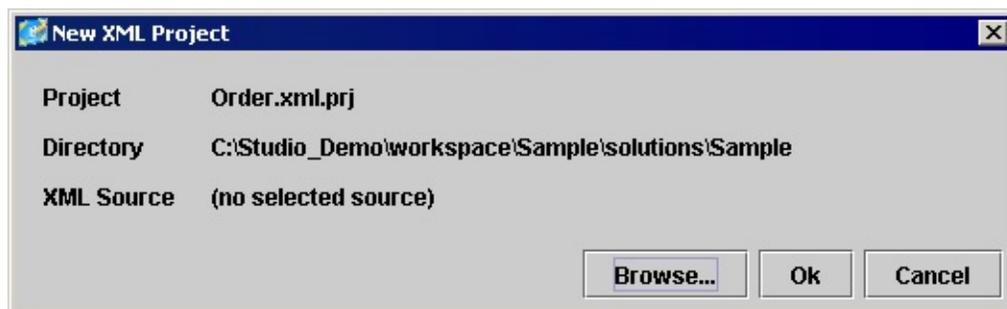
The project file only contains references to marked XML elements and attributes. The external sample XML document is read every time the project is opened. If in the future the XML document requires new elements or attributes just replace the existing sample XML document and these new elements or attributes will appear in the view.

7.21.3 Create an XML Binding Project

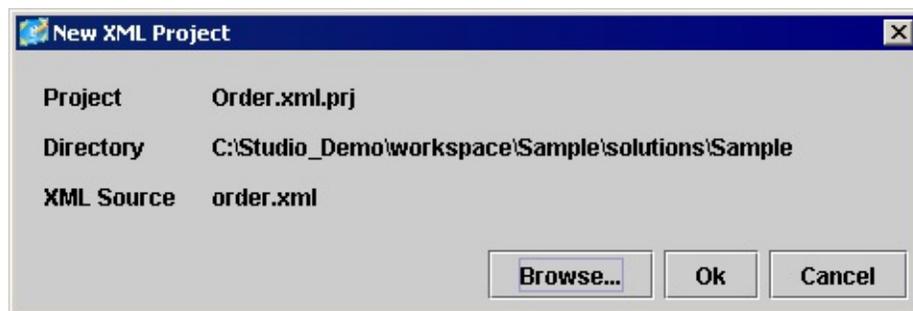
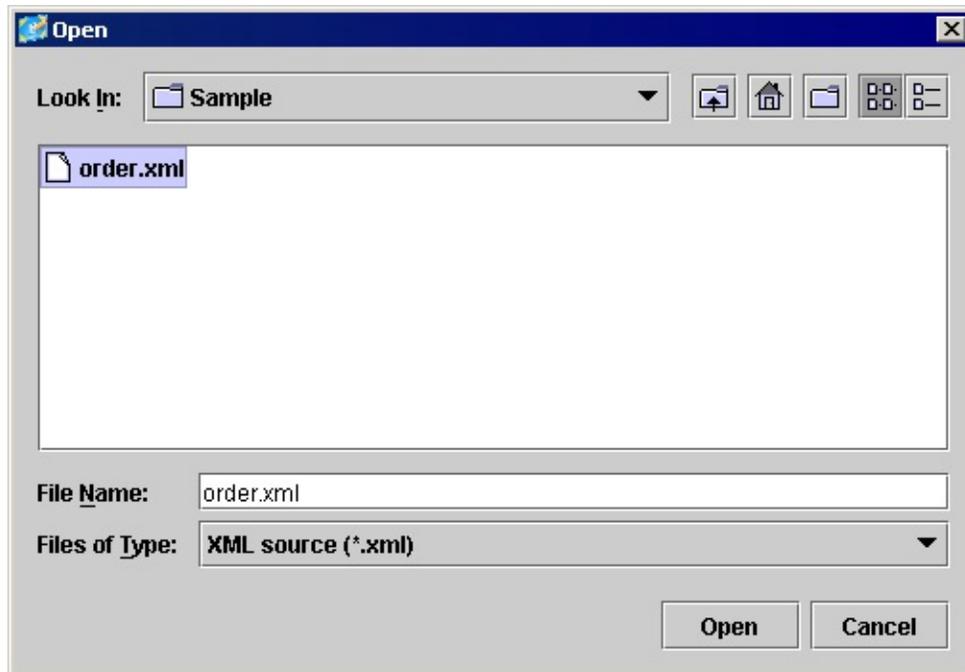
1. From the Project Solutions node select the *New Solution* menu item.
2. Select the *XML Binding Wizard* tool and enter or select the *Group* folder to receive the project file.
3. Enter the project File name. The extension '.xml.prj' is added automatically.



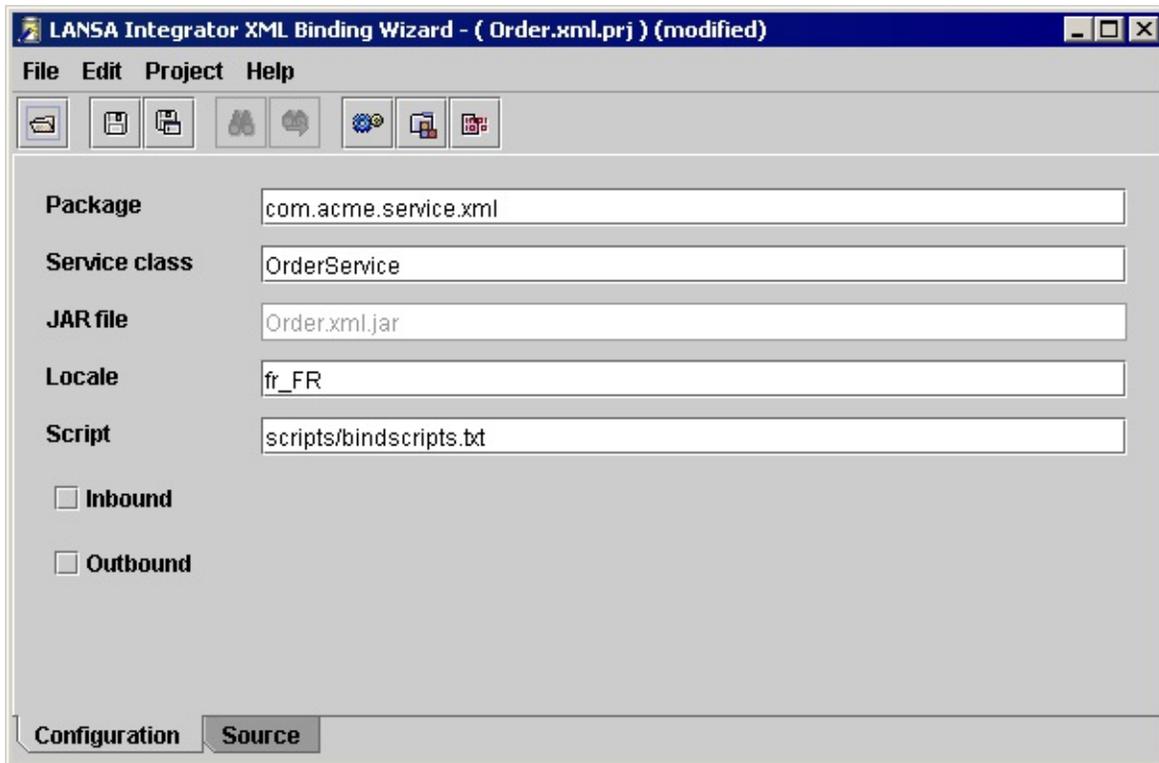
4. Press the OK button to continue.



5. Press the browse button to locate the sample XML file.



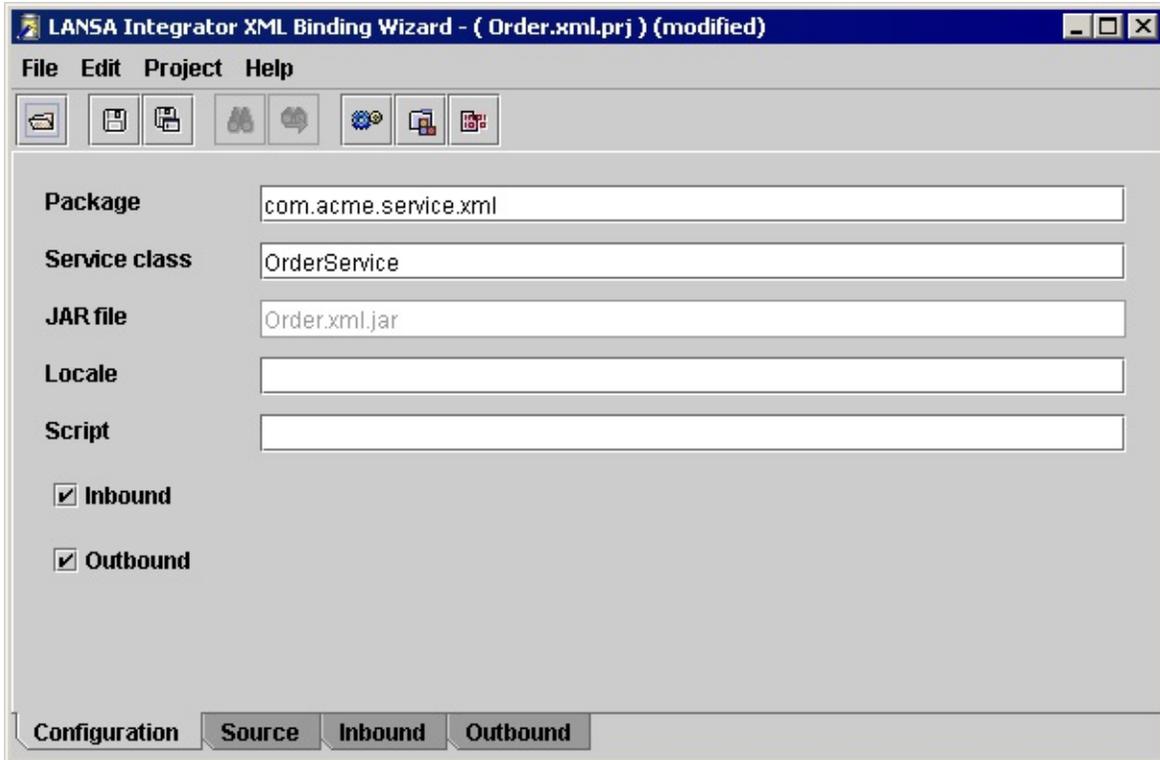
6. Press the OK button to create the project.



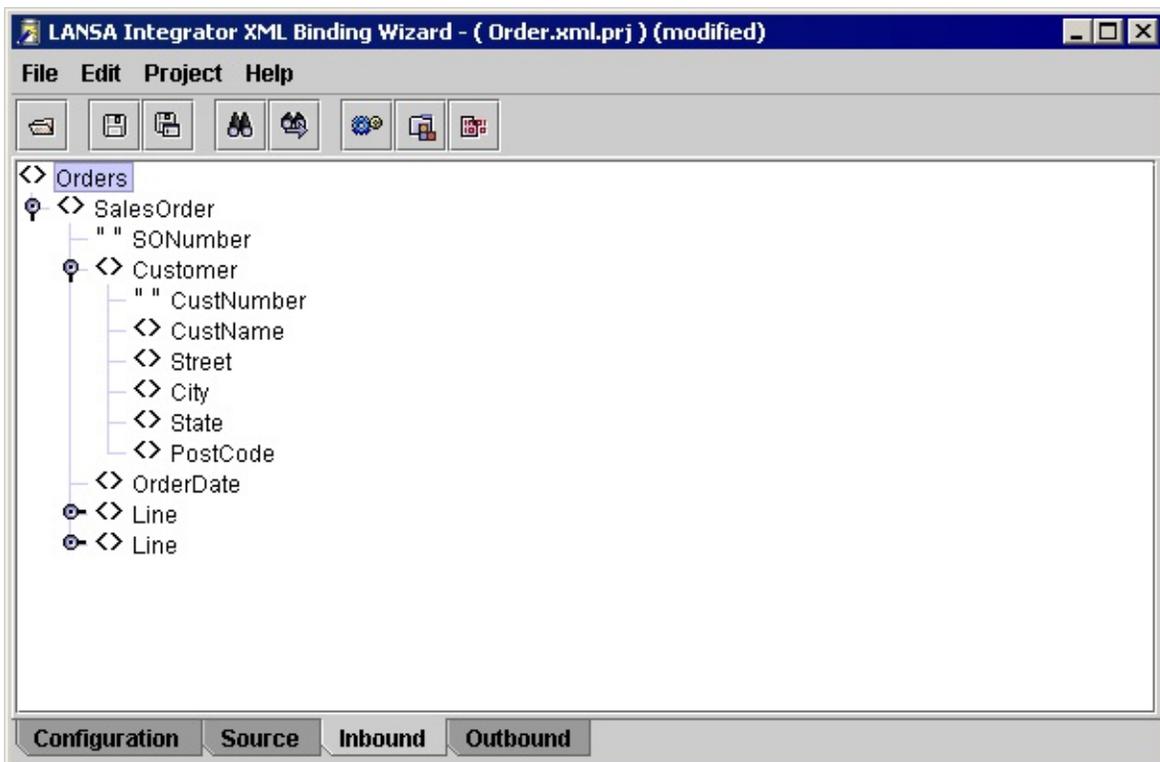
7. Select whether the project is for inbound mapping, outbound mapping or both.

Normally the Locale field is left blank, but if an XML element or attribute contains numbers formatted in a particular locale then enter this locale and check the Number Format option against this element or attribute.

If value formatting is going to be done using JavaScript functions, then the JavaScript file containing the functions needs to be specified in the Script field. Refer to [JavaScript Function Example](#) for notes on doing this.



8. Select the Inbound tab to map element and attributes.



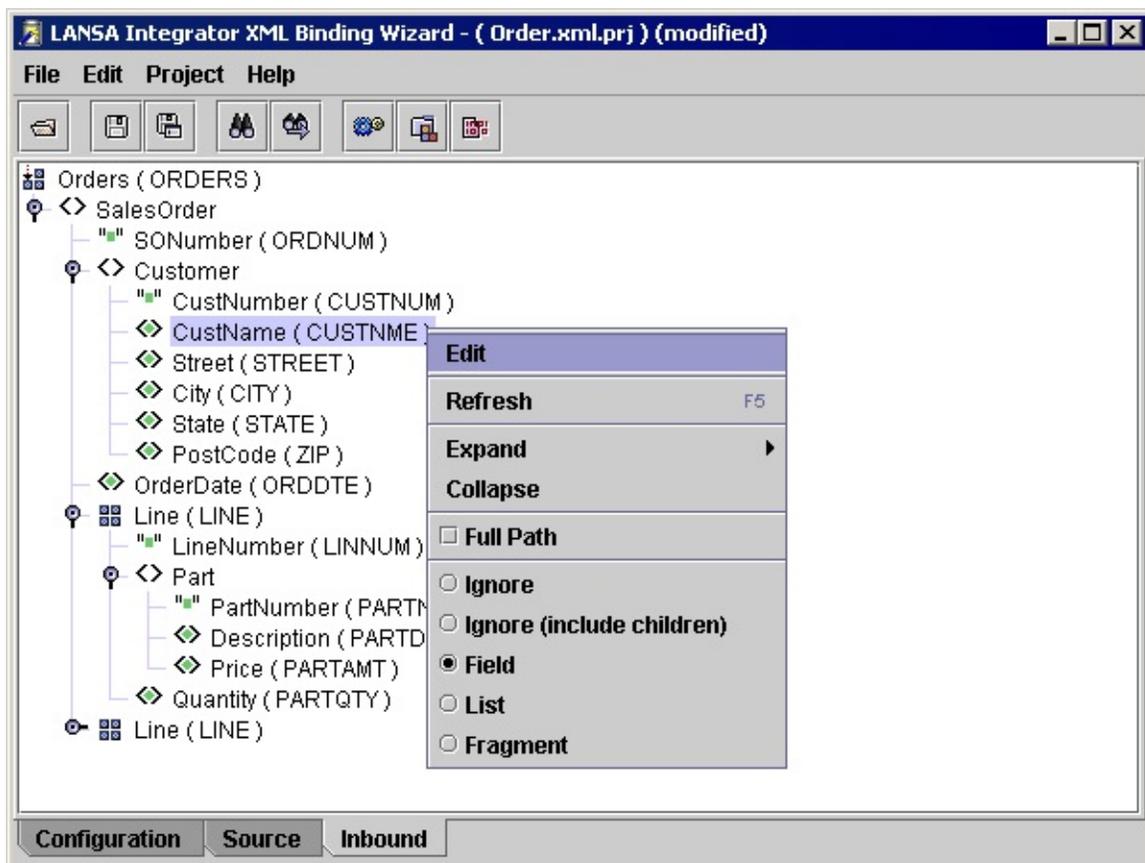
The element and attribute nodes can be marked to accept and handle data in

different ways.

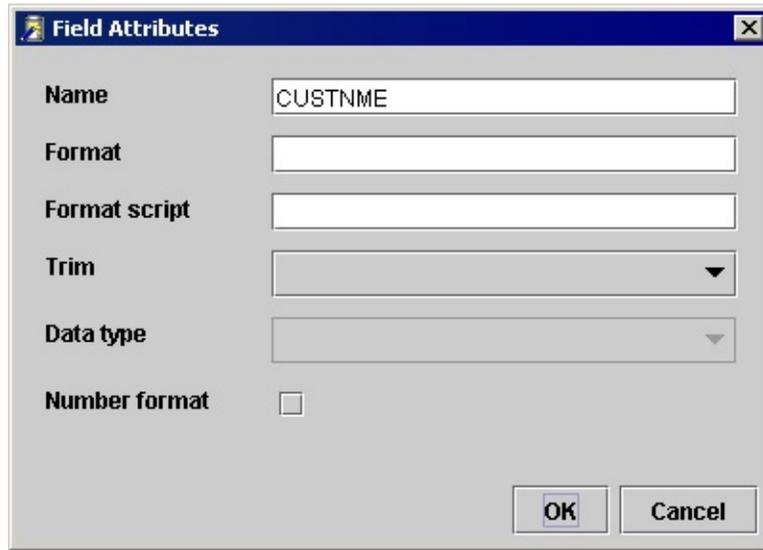
Different icons indicate the node types:

- <> Element
- ◊ Element selected
- ◊ Element selected and has a field name
- " " Attribute
- " " Attribute selected
- " " Attribute selected and has a field name
- ☐ Element marked as a List
- ☐ Element marked as a Fragment

9. Right mouse click on the selected node to display the Pop-up Menu. The menu items enabled depend on the node selected. You can also use the [7.24 Repository Viewer](#) to drag and drop a LANSAs field name onto a node.

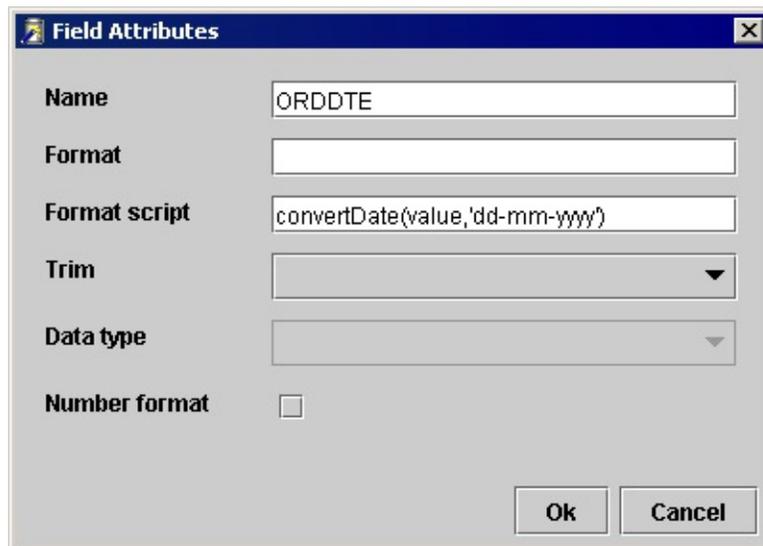


10. Press the Enter key or the Edit menu item to assign a binding field.



11. If the XML value needs to be formatted using a JavaScript function, enter the function and the value keyword. The value keyword instructs the JavaScript Engine to use the XML value. It is possible to pass a static value as a parameter.

In the following example a format pattern is being passed. Embedded blanks can exist in the function script, but any static values need to immediately follow the comma for successful parsing of the parameters.



12. Press the build button to generate and compile the service, this service class will be added to the jar file.

JavaScript Function Example

The following example illustrates how a JavaScript function is written. The function declaration, function name and parameters enclosed with brackets must be on a single line.

Multiple functions can exist in the same source file.

```
function concat ( value1, value2 )
{
    return value1 + value2 ;
}
```

```
function startDate ( type )
{
    if ( type == 'A' )
    {
        return "30-1-2006" ;
    }

    return "23-4-2005" ;
}
```

JavaScript is a loosely typed language.

That does not mean that it has no data types, just that the value of a variable does not need to have a particular type of value assigned to it nor does it need to always hold the same type of value.

JavaScript will freely type-convert values into a type suitable for (or required by) the context of the variable's use.

JavaScript being loosely typed and willing to type-convert still does not save you from needing to think about the actual types of values that you are dealing with.

A problem arises from the dual nature of the + operator used for both numeric addition and string concatenation. The nature of the operation performed is determined by the context. If both operands are numbers to start with, the + operator performs addition, otherwise it converts all of its operands to strings and does concatenation.

The mapping engine passes all values to the JavaScript function as String data

types.

It is best to convert parameter values to their required data types.

There are several JavaScript functions such as `parseFloat`, `parseInt` and the `Number` object that can be used to convert `String` values to number values. Also number variables have several built-in functions such as `toFixed` and `toPrecision` that allow formatted values to be returned.

```
value = Number ( value )  
value = parseInt ( value )  
value = parseFloat ( value )
```

The following example illustrates these functions.

```
function changeit ( value )  
{  
    value = Number ( value );  
  
    if ( value > 100 )  
    {  
        value = value - 100 ;  
    }  
  
    return value.toFixed ( 2 );  
}
```

7.22 JSON Binding Wizard

The JSON Binding Wizard is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The JSON Binding Wizard is an application that allows you to map JSON object fields to LANSAs fields and lists. The Wizard will generate code to be used by JSON binding services to read and write JSON objects.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. Refer to <http://www.json.org/>.

AJAX (Asynchronous JavaScript And XML) is a web development technique used for creating interactive web applications. The intent is to make web pages feel more responsive by exchanging small amounts of data with the server behind the scenes, so that the entire web page does not have to be reloaded each time the user requests a change. This is intended to increase the web page's interactivity, speed, functionality, and usability.

The JSON Binding Wizard requires a Java compiler, such as Oracle's Java SDK, to be installed on the PC.

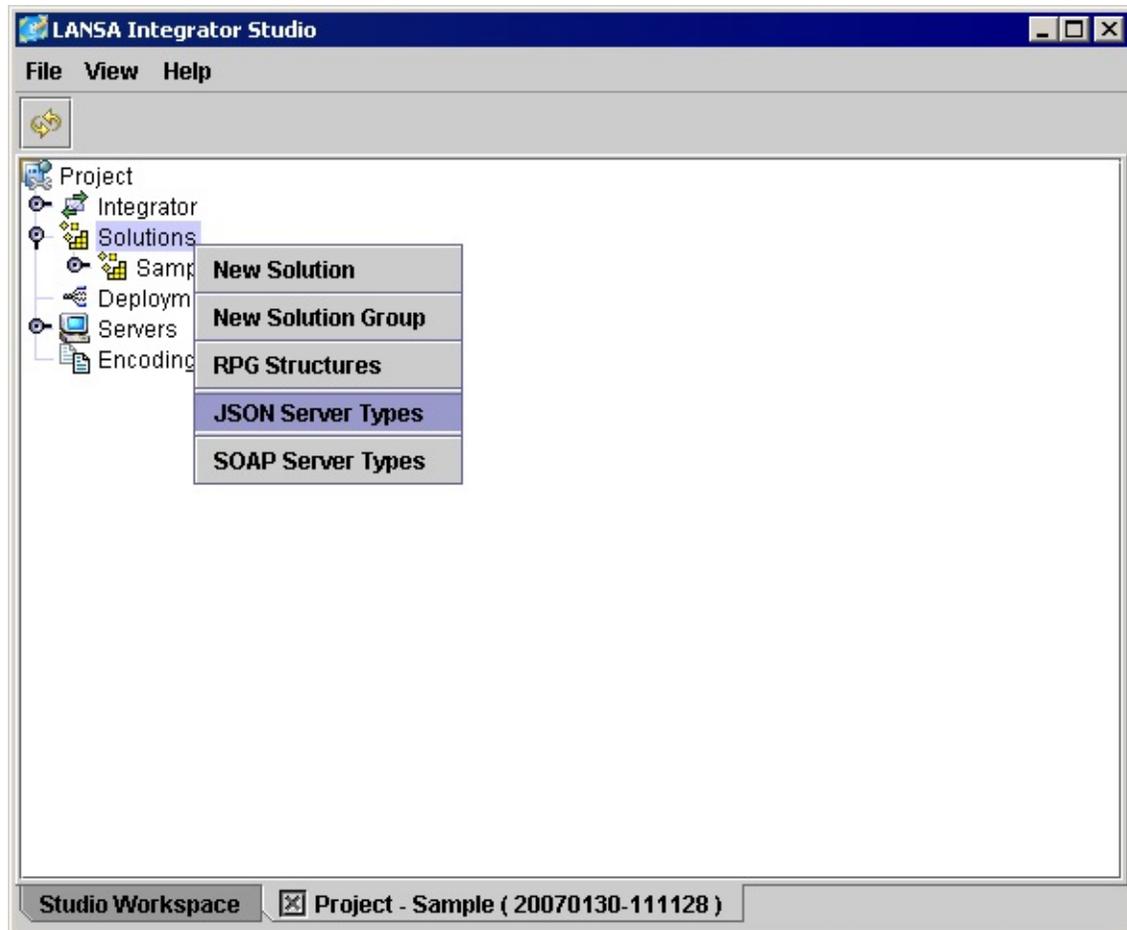
Refer to [7.1.2 Set Java Compiler and Java Runtime](#).

Review the following topics:

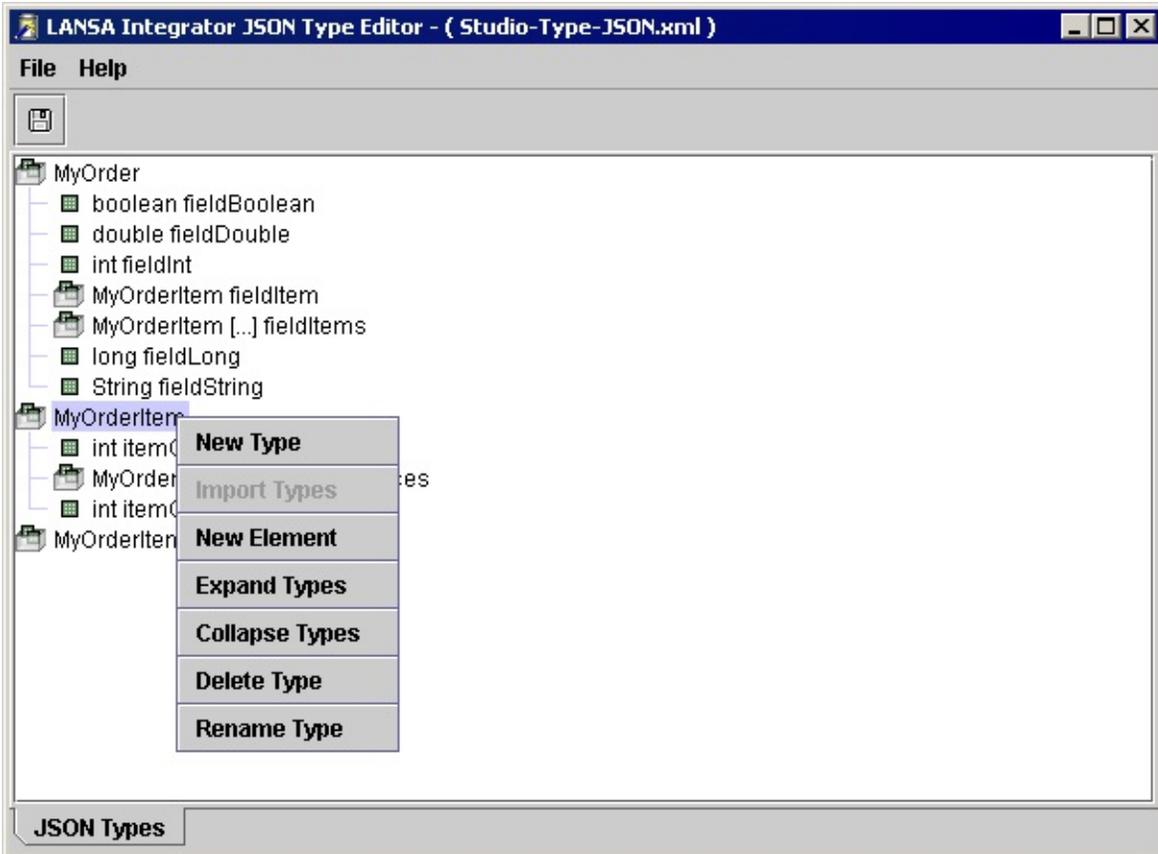
- [7.22.1 Create a JSON Type](#)
- [7.22.2 Create a JSON Binding Project](#)
- [7.22.3 Create a JSON HTML Browser Client](#)

7.22.1 Create a JSON Type

From the Solution folder pop-up menu, select the *JSON Types* menu item.

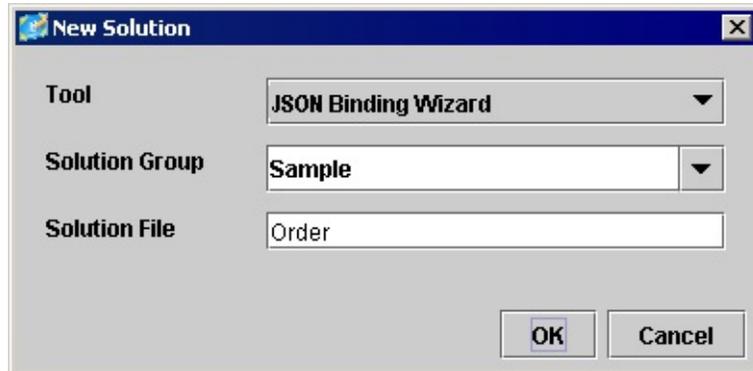


The JSON Type Editor allows types to be created and maintained.

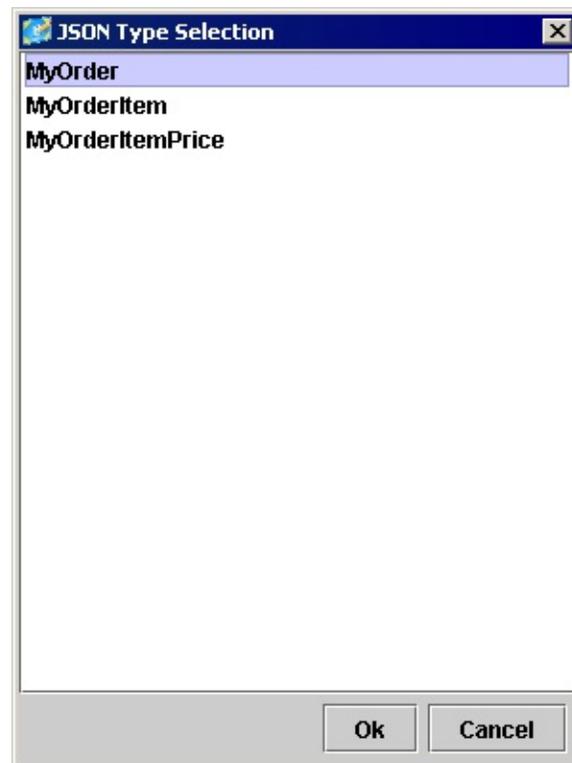


7.22.2 Create a JSON Binding Project

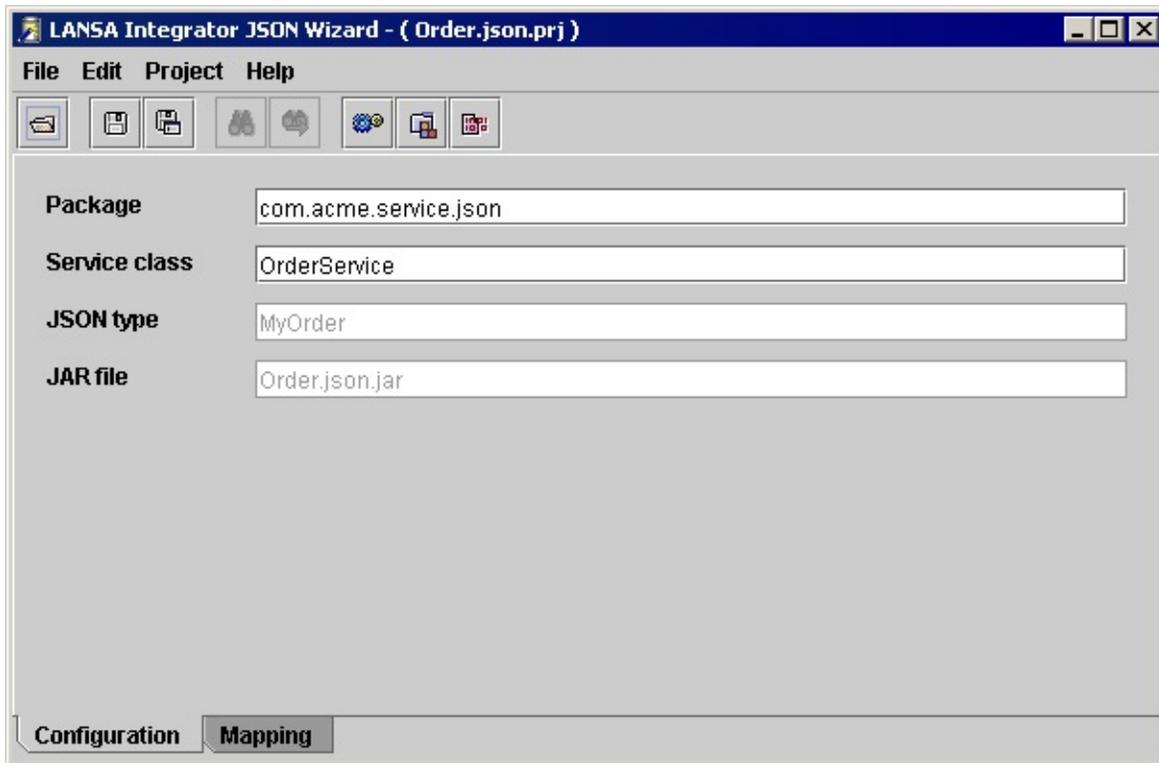
1. From the Project Solutions node select the *New Solution* menu item.
2. Select the *JSON Binding Wizard* tool and enter or select the *Group* folder to receive the project file.
3. Enter the project File name. The extension '.json.prj' is added automatically.



4. Press the OK button to continue.
5. Select the JSON Type that will be used for the project mapping.

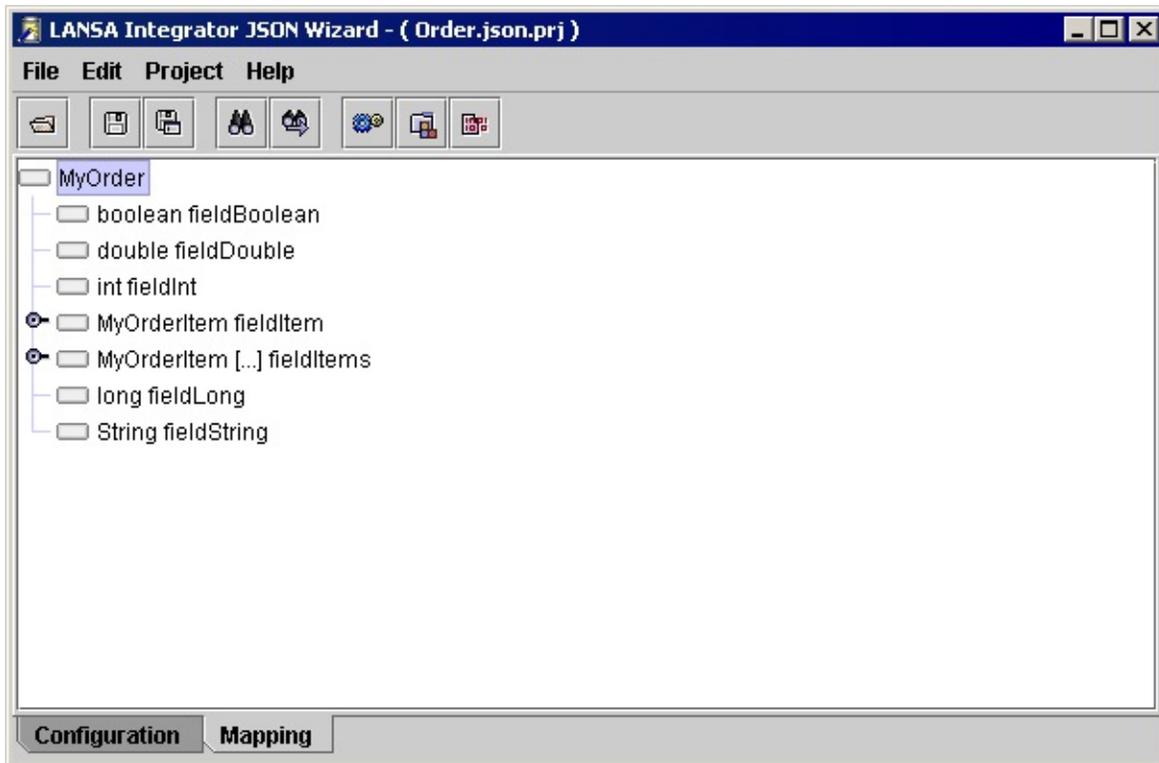


6. Press the OK button to create the project.



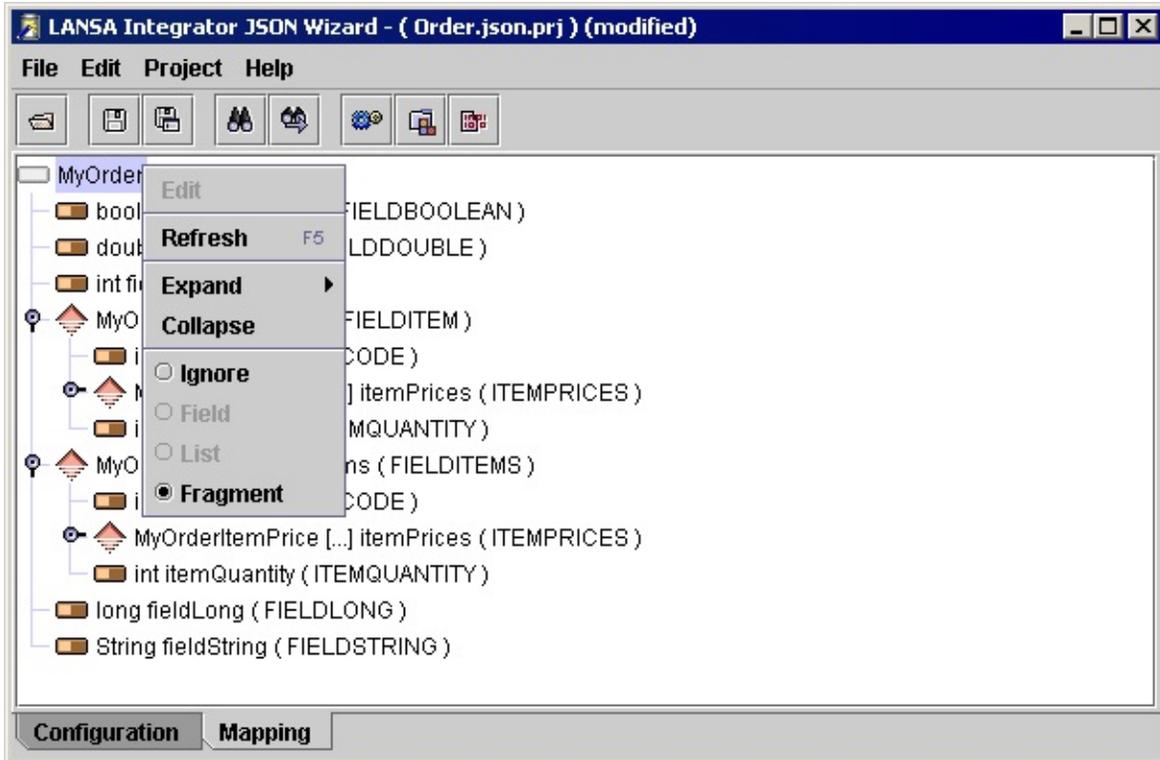
The project name is used in the creation of the JAR file name and the service class name.

7. Select the Mapping tab to maintain the mapping.

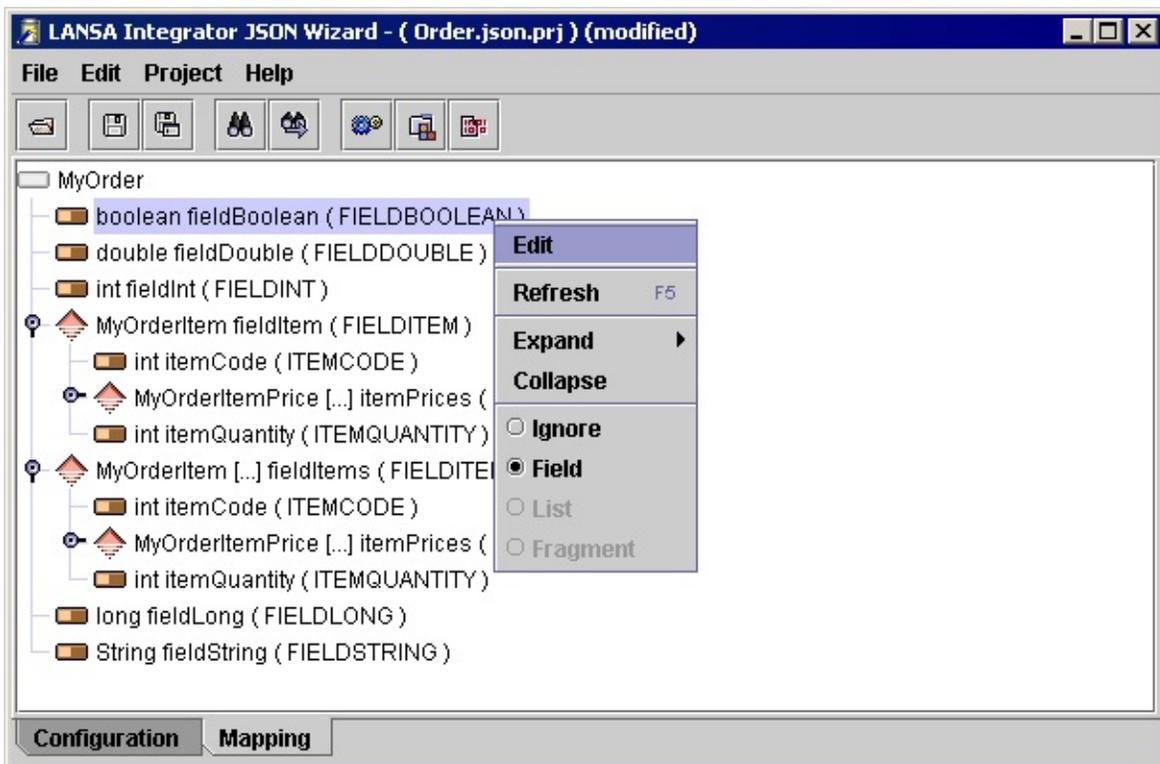


8. On the root level node select the Fragment menu item and all child nodes will be automatically marked.

The root level node is not marked as a fragment, as it is an implied fragment and when the BIND command executes the first level child nodes of the root node will bind to the function.



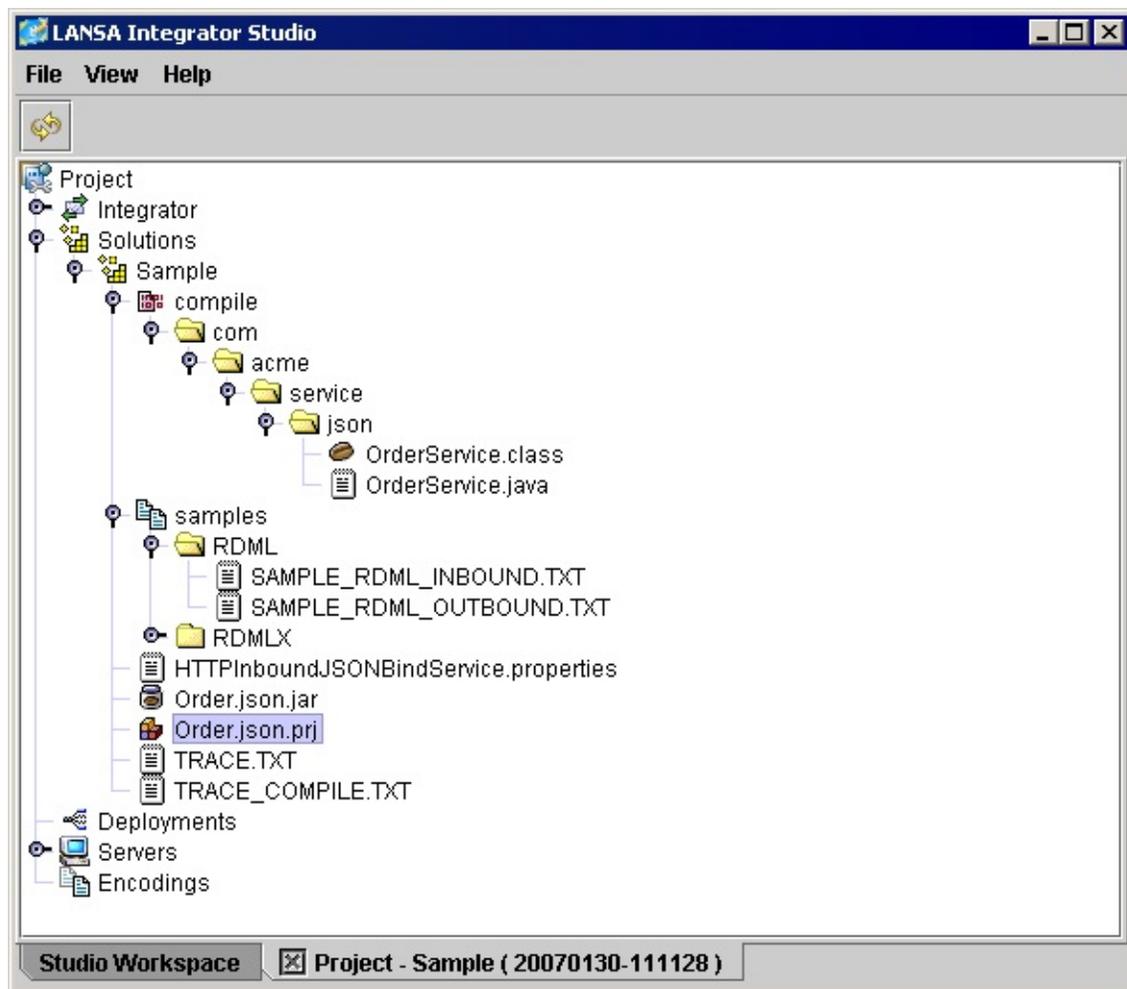
- Assign program fields to the JSON object fields. You can also use the [7.24 Repository Viewer](#) to drag and drop a LANSA field name onto a node.



10. Press the build button to generate and compile the service. The service class will be added to the jar file.



All generated code and compiled classes exist in the compile sub directory. Files containing service properties are created in the solution directory and samples are created in the samples sub directory.



7.22.3 Create a JSON HTML Browser Client

The following HTML illustrates how to send and receive a JavaScript object using a browser client.

The JavaScript object needs to be created using the same field elements as the JSON Type.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml-strict.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
```

```
<meta http-equiv="content-type" content="text/html; charset=utf-8">
```

```
<title>JSON AJAX Sample</title>
```

```
<script type="text/javascript" src="json2.js"></script>
```

```
<script type="text/javascript" src="jsonajax.js"></script>
```

```
<script type="text/javascript">
```

```
function sendOrder ()
```

```
{
```

```
    /*
```

```
        Get some input and reset form inputs
```

```
    */
```

```
    var myInput = document.getElementById ( "ID_1" ).value ;
```

```
    document.getElementById ( "ID_1" ).value = "" ;
```

```
    document.getElementById ( "ID_2" ).value = "" ;
```

```
    document.getElementById ( "ID_3" ).value = "" ;
```

```
    /*
```

```
        Create MyOrder object
```

```

*/

var myOrder = new Object () ;

myOrder.fieldString = myInput ;
myOrder.fieldBoolean = true ;
myOrder.fieldDouble = 24.56 ;
myOrder.fieldInt = 987 ;
myOrder.fieldLong = 123456 ;

var myOrderItem1 = new Object () ;
myOrderItem1.itemCode = 231 ;
myOrderItem1.itemQuantity = 25 ;

var myOrderItemPrice11 = new Object () ;
myOrderItemPrice11.priceAmount = 10.67 ;
myOrderItemPrice11.priceType = "PT_A" ;

var myOrderItemPrice12 = new Object () ;
myOrderItemPrice12.priceAmount = 10.67 ;
myOrderItemPrice12.priceType = "PT_A" ;

myOrderItem1.itemPrices = new Array ( myOrderItemPrice11, myOrderItemPrice12 ) ;

myOrder.fieldItem = myOrderItem1 ;

myOrder.fieldItems = new Array ( myOrderItem1, myOrderItem1 ) ;

/*
    Send MyOrder
*/

var session = new JSONSession ( "http://lansa01:1099/cgi-bin/jsmpoxy?
json" ) ;

// var responseObject = session.send () ;
// var responseObject = session.receive () ;

var responseObject = session.send ( myOrder ) ;

```

```
if ( responseObject == null )
{
    alert ( session.getStatus () );

    alert ( session.getResponseText () );

    return ;
}

/*
    Update form inputs with response object values
*/

document.getElementById ( "ID_1" ).value = responseObject.fieldString ;
document.getElementById ( "ID_2" ).value = responseObject.fieldDouble ;
document.getElementById ( "ID_3" ).value = responseObject.fieldInt ;
}

</script>

</head>

<body>

<form>

    <input type="text" size="15" id="ID_1"><p/>
    <input type="text" size="15" id="ID_2"><p/>
    <input type="text" size="15" id="ID_3"><p/>
    <input type="button" value="Send Order" onClick="sendOrder()">

</form>

</body>

</html>

jsonajax.js
```

```
/*jslint browser: true, undef: true, nomen: true, eqeqeq: true, strict: true */
```

```
/*global window, ActiveXObject */
```

```
"use strict";
```

```
function JSONSession ( endpoint )
```

```
{
```

```
    var m_xhr = null ;
```

```
    var m_endpoint = endpoint ;
```

```
    /*
```

```
        Private functions
```

```
    */
```

```
function createXMLHttpRequest ()
```

```
{
```

```
    if ( m_xhr === null )
```

```
    {
```

```
        if ( window.XMLHttpRequest )
```

```
        {
```

```
            /*
```

```
                IE8, IE7, Firefox, Chrome, Safari, Opera
```

```
            */
```

```
            m_xhr = new XMLHttpRequest () ;
```

```
            return true ;
```

```
        }
```

```
        if ( window.ActiveXObject )
```

```
        {
```

```
            /*
```

```
                IE6
```

```
            */
```

```
            m_xhr = new ActiveXObject ( "Msxml2.XMLHTTP" ) ;
```

```
        return true ;
    }

    return false ;
}

return true ;
}

function doGET ()
{
    try
    {
        m_xhr.open ( "GET", m_endpoint, false ) ;

        m_xhr.setRequestHeader ( "Connection", "close" ) ;

        m_xhr.send ( null ) ;

        if ( m_xhr.readyState !== 4 )
        {
            return false ;
        }

        if ( m_xhr.status === 200 )
        {
            return true ;
        }

        return false ;
    }
    catch ( e )
    {
        alert ( "doGET: " + e ) ;

        return false ;
    }
}
```

```
function doPOST ( content )
{
    try
    {
        m_xhr.open ( "POST", m_endpoint, false );

        m_xhr.setRequestHeader ( "Content-Type", "application/json" );
        m_xhr.setRequestHeader ( "Connection", "close" );

        m_xhr.send ( content );

        if ( m_xhr.readyState !== 4 )
        {
            return false ;
        }

        if ( m_xhr.status === 200 )
        {
            return true ;
        }

        return false ;
    }
    catch ( e )
    {
        alert ( "doPOST: " + e );

        return false ;
    }
}
```

```
function isJSON ()
{
    var value = m_xhr.getResponseHeader ( "Content-Type" );

    if ( value === null )
    {
        return false ;
    }
}
```

```
/*
    application/json
*/

if ( value.length < 16 )
{
    return false ;
}

value = value.substring ( 0, 16 ).toLowerCase () ;

if ( value === "application/json" )
{
    return true ;
}

return false ;
}

function parseJSON ()
{
    try
    {
        return JSON.parse ( m_xhr.responseText ) ;
    }
    catch ( e )
    {
        var date = new Date () ;

        var name = "error" + date.getTime () ;

        var error = window.open ( "",
                                name,
                                "width=800,height=600,menubar=0,toolbar=0,status=0"

        error.document.title = "JSON parse error" ;

        var element = error.document.createElement ( "pre" ) ;
```

```

        element.appendChild ( error.document.createTextNode ( m_xhr.responseText ) );
        error.document.body.appendChild ( element ) ;

        return null ;
    }
}

/*
    Public functions
*/

this.send = function ( objectSend )
{
    if ( !createXMLHttpRequest () )
    {
        alert ( "Cannot create XMLHttpRequest object" ) ;

        return null ;
    }

    var content = null ;

    if ( objectSend === undefined || objectSend === null )
    {
        /*
            If the send function is called with no parameters, then objectSend is undefined
        */

        content = "" ;
    }
    else
    {
        content = JSON.stringify ( objectSend ) ;
    }

    if ( !doPOST ( content ) )
    {

```

```
        return null ;
    }

    if ( isJSON () )
    {
        return parseJSON () ;
    }

    return null ;
};

this.receive = function ()
{
    if ( !createXMLHttpRequest () )
    {
        alert ( "Cannot create XMLHttpRequest object" ) ;

        return null ;
    }

    if ( !doGET () )
    {
        return null ;
    }

    if ( isJSON () )
    {
        return parseJSON () ;
    }

    return null ;
};

this.getStatus = function ()
{
    if ( m_xhr === null )
    {
        return 0 ;
    }
}
```

```
    return m_xhr.status ;
};

this.getResponseText = function ()
{
    if ( m_xhr === null )
    {
        return null ;
    }

    return m_xhr.responseText ;
};

this.getXMLHttpRequest = function ()
{
    return m_xhr ;
};
}
```

7.23 XML Transformation Wizard

The XML Transformation Wizard is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The XML Transformation Wizard is an application that allows you to map the XML elements and attributes to LANSAs fields and lists. The Wizard will automatically create style sheets that can be used to transform inbound XML to LANSAs fields and lists, or to transform LANSAs fields and lists to outbound XML. The XML Transformation Wizard removes the LANSAs developer from the complexities of creating style sheets to perform the required XML transformations.

Review the following topics:

[7.23.1 Architecture Overview](#)

[7.23.2 Sample XML Document](#)

[7.23.3 Create an XML Transformation Project](#)

[7.23.4 How to use the XML Transformation Wizard](#)

7.23.1 Architecture Overview

A transformation involves the mapping of the content of XML documents to and from LANSAs fields or lists. It requires an XML document and a style sheet. The style sheet defines the rules of processing or transforming the XML.

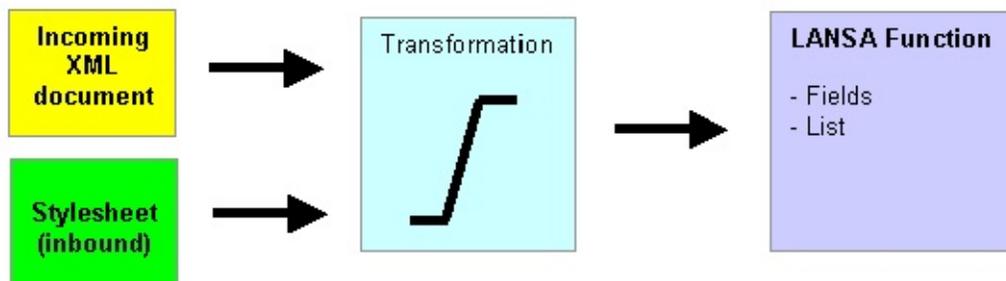
The purpose of the XML Transformation Wizard is to create the style sheets based on XML documents and the required fields or lists in your LANSAs functions.

Depending on the application you are developing you may need to define style sheets to process either incoming XML documents or to generate outgoing XML documents, or both.

Note: The XML Transformation Wizard is only used to generate and test the style sheets. These style sheets must be manually copied to the Server where the JSM Service is running. By default, the style sheets are stored in the /jsm/instance/xsl directory.

Style Sheets to process incoming XML documents

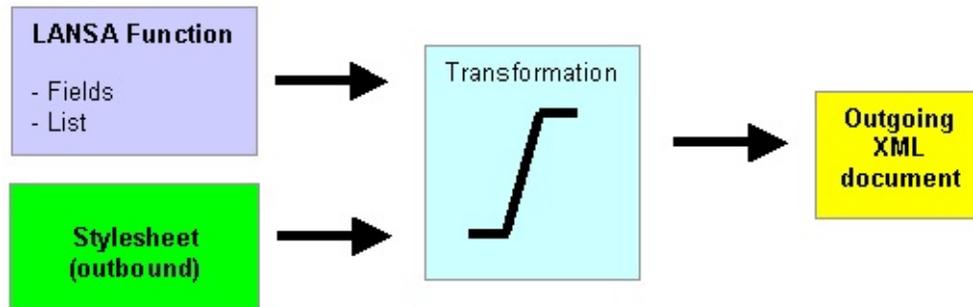
An incoming XML document is an XML document sent to your application. The XML to be used is normally defined by the sending application. The XML Transformation Wizard generates the style sheet to process the incoming XML document and map its content into LANSAs structures such as function fields and lists. The generated style sheet is displayed in the XML Transformation Wizard's Inbound Transformation tab.



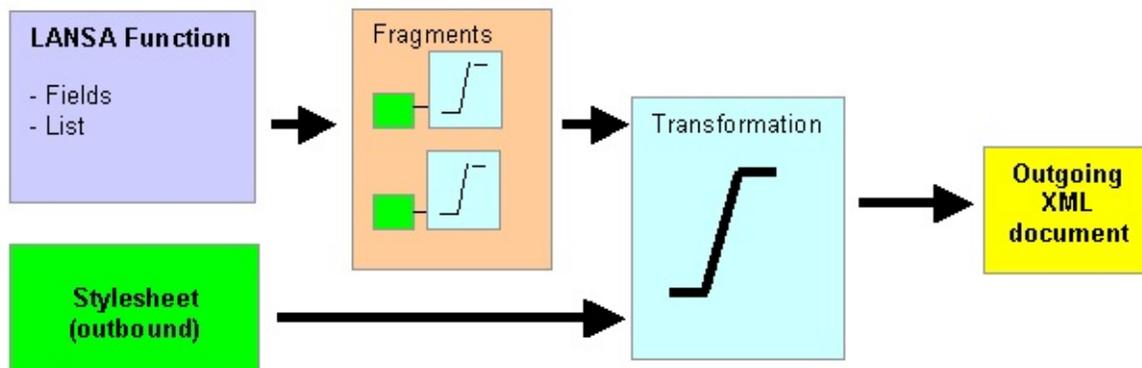
Style Sheets to generate outgoing XML documents

An outgoing XML document is an XML document you send to another application. The XML to be used is normally defined by the receiving application, i.e. you must format your outgoing document to match their XML

standard. The XML Transformation Wizard generates the style sheet to generate the outgoing XML document from data provided by LANSAs structures such as function fields and lists. The generated style sheet is displayed in the XML Transformation Wizard's Outbound Transformation tab.



To generate more complex outgoing XML documents you may want to work with multiple lists and therefore apply multiple transformations. In this case the XML Transformation Wizard talks about Fragments. Each fragment is generated/transformed individually out of LANSAs structures and merged together by the style sheet displayed in the XML Transformation Wizard's Outbound Transformation tab.



7.23.2 Sample XML Document

The XML Transformation Wizard requires a sample XML document. It does not directly use an XML Schema or DTD file.

It is imperative that your sample XML document contains all XML elements and attributes required for mapping.

The sample XML document must be well formed and follow the rules and syntax of XML documents.

The sample XML file may be provided by a third party application or you may need to create your own XML document.

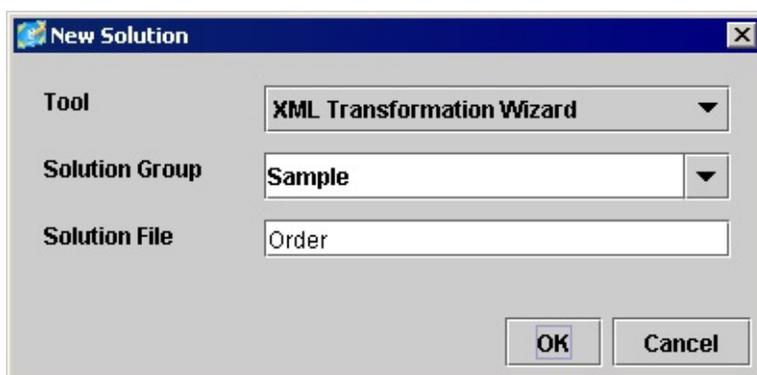
You can create a sample XML file using a text editor or use the [7.12 XMLSchema Viewer](#) to generate a sample XML file from an XML Schema file.

7.23.3 Create an XML Transformation Project

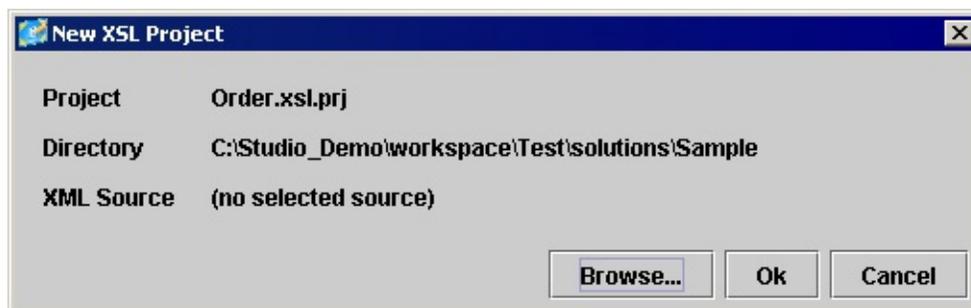
The first step in transforming an XML document is to create a new project. You can do this either by selecting File New from the XML Transformation Wizard menu or create a new Studio Solution.

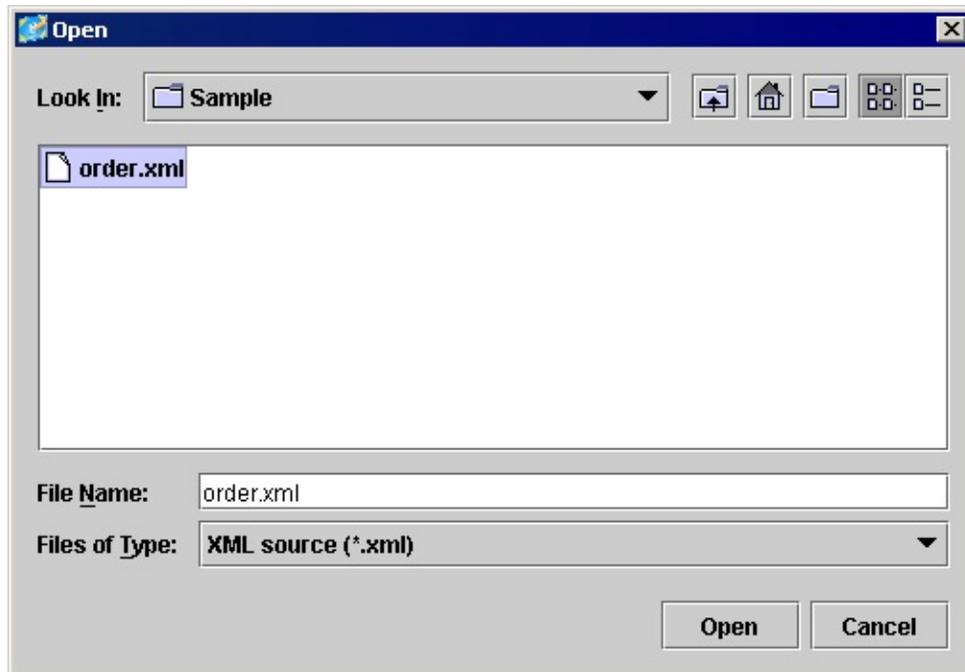
It is recommended that you create a new project for each transformation in order to help organize the information in your application.

1. From the Project Solutions node select the *New Solution* menu item.
2. Select the *XML Transformation Wizard* tool and enter or select the *Group* folder to receive the project file and sample XML document.
3. Enter the project File name. The extension *'.xsl.prj'* is added automatically.

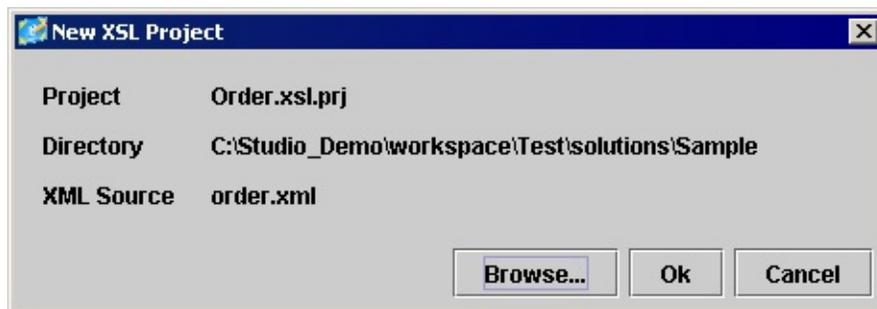


4. Press the *OK* button and then browse the local file system for the sample XML document source.





5. Select the sample XML document and press the OK button, the sample XML document will be copied into the group folder and an XML Transformation Wizard project file will also be created.



7.23.4 How to use the XML Transformation Wizard

How to use the Wizard is described in the following topics:

[File Menu](#)

[Project Tab](#)

[Source Tab](#)

[Fields Tab](#)

[List Tab](#)

[Inbound Transformation Tab](#)

[Outbound Transformation Tab](#)

[Map Fields](#)

[Test Transformation](#)

[Save Transformation](#)

File Menu

The file menu commands are:

New

A project file is used to store the mapping information used to build the Inbound / Outbound XSL style sheets.

When the New option is selected, a standard file dialog is displayed allowing you to select the sample XML document for this new project.

Open

To open an existing project select the Open menu command. The Open dialog will then be displayed allowing you to select the project file you want to open. The project file that you select will initially be displayed in the [Project Tab](#).

If you have an existing project open, be sure to save this project before opening another project.

Close

Closes the project without saving the changes.

Save

Saves the current project under the current name.

Save As

If you select Save As you can save the current project under a different project name.

Preference

Allows wizard preferences to be set and saved to the current project file.

- Select Use 3GL structures to enable field structure information and structure XML creation.
- Select Indent Outbound XML to have `<xsl:output indent="yes" xalan:indent-amount="0"/>` included in the generated outbound XML.
- Select Omit XML declaration if `<?xml version="1.0" encoding="UTF-8"?>` is not to be included in the generated outbound XML.



About

Displays the XML Transformation Wizard version information and copyright message.

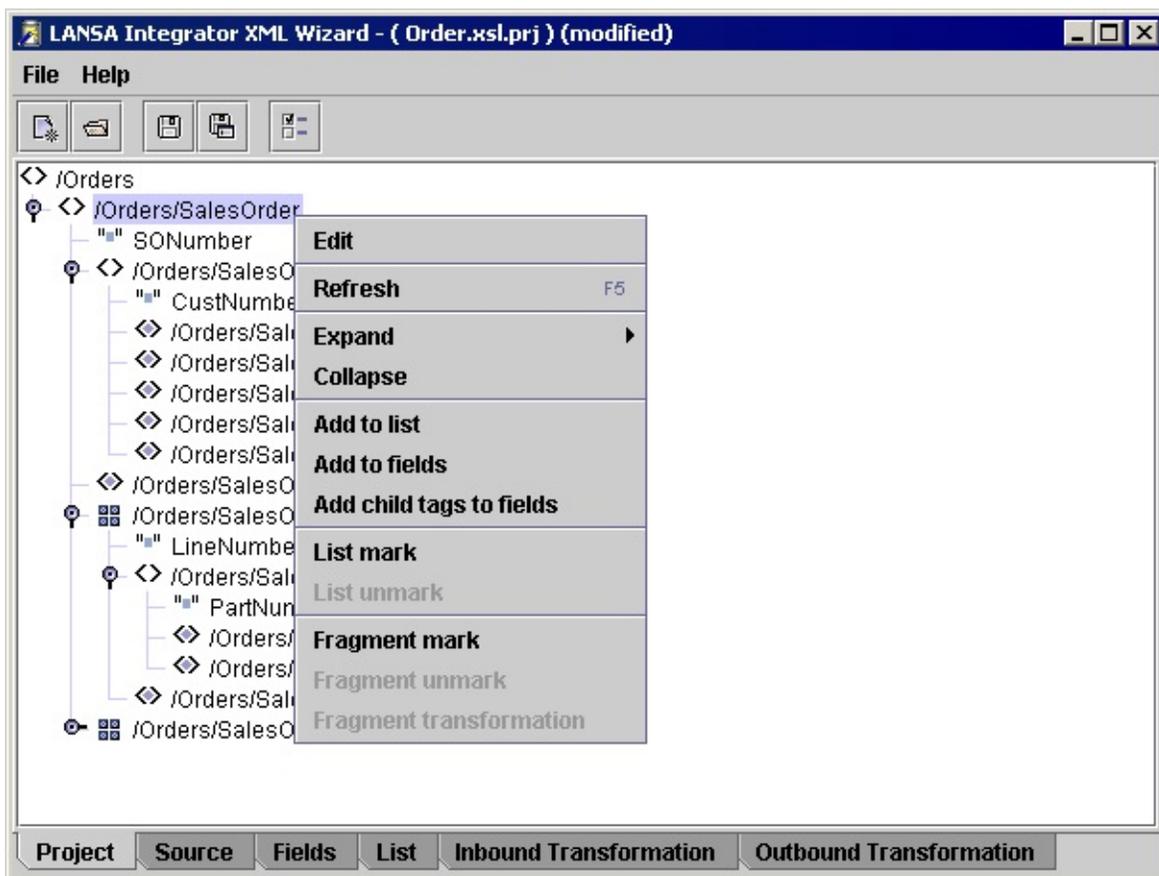
Exit

Exits the XML Transformation Wizard.

Note: To save the changes made to the project, select Save or Save As before exiting from the XML Transformation Wizard.

Project Tab

When you open a project a graphical representation of the components that contribute to the construction of Inbound and Outbound Transformation style sheets is displayed.



Different icons indicate the node types:

- ◁▷ Element
- ◁▷ Element selected
- ◁▷ Element selected and has a field name
- " " Attribute
- " " Attribute selected
- " " Attribute selected and has a field name
- ■ Element marked as a List

Element marked as a Fragment

Right mouse click on the selected node to display the Pop-up Menu. The menu items enabled depend on the node selected.

Edit

Allows the RDML field name to be assigned to this element or attribute. Pressing the enter key has the same affect.

Add to list

Add the selected node to the field list collection. Candidates for this kind of selection are elements or attributes that occur more than once in the XML. To edit the list, refer to the [List Tab](#).

Add to fields

Adds the selected node to the field collection. Candidates for this kind of selection are elements or attributes that occur only once in the XML. To edit the fields, refer to the [Fields Tab](#).

The Insert key performs the same function.

Note: The program will not add entries to the Fields collection if the entries are already assigned to the List collection.

Add child tags to fields

Adds all child tags and attributes of the selected node to the current collection of fields. This collection can be maintained from the [Fields Tab](#) or the [Project Tab](#). From the Fields tab it is possible to delete entries.

It is recommended that you mark the list first before adding all child tags to the field collection.

Note: The program will not add entries to the Fields collection if the entries are already assigned to the List collection.

List mark

Will add all child tags and attributes of the selected node to the field list collection. This collection can be maintained from the [List Tab](#) or the [Project Tab](#). From the List tab it is possible to delete entries.

Note: that the program will not add entries to the List collection if the entries are already assigned to the Fields collection.

List unmark

This unmarks the selected list node and clears the field List collection.

Fragment mark

This marking allows an XSL style sheet based on the selected node and its sub-nodes to be created and saved to a separate file. This style sheet will then be used to do the transformation of the selected Fragment. The output produced by this transformation will then be merged with the final XML document according to the definitions in the XSL from the Outbound Transformation tab. This XSL will include references to the XSL Fragments.

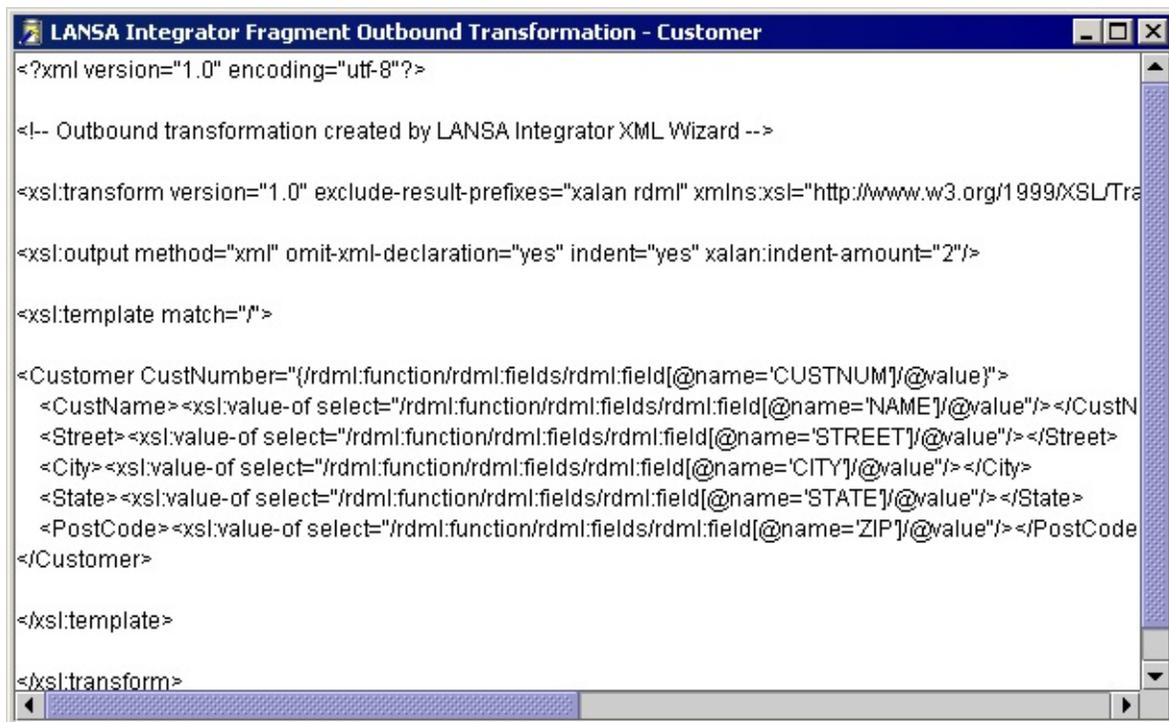
Fragments allow the Outbound XML to be broken up into smaller components, which can be assembled to create a more complex XML document.

Fragment unmark

Removes the fragment mark.

Fragment transformation

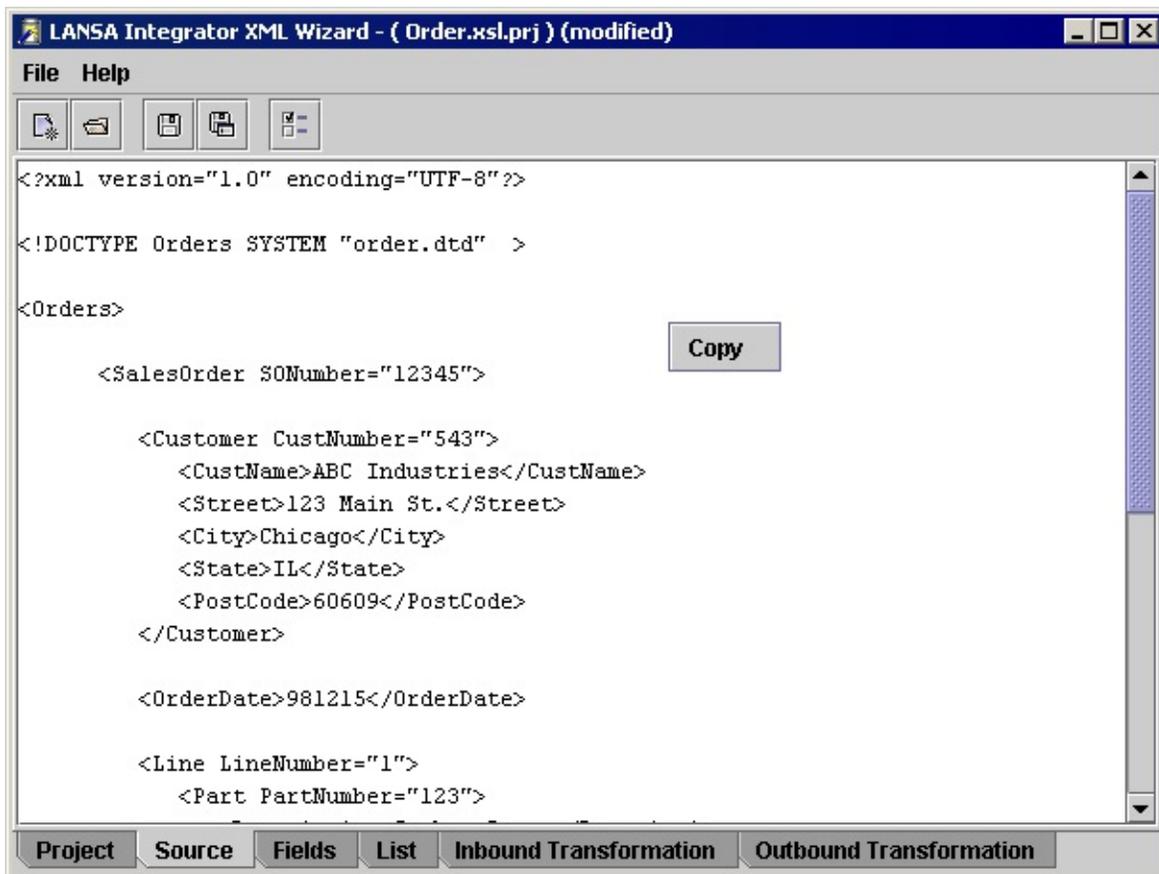
Creates the style sheet for this fragment in a separate window:



```
<?xml version="1.0" encoding="utf-8"?>
<!-- Outbound transformation created by LANSA Integrator XML Wizard -->
<xsl:transform version="1.0" exclude-result-prefixes="xalan rdml" xmlns:xsl="http://www.w3.org/1999/XSL/Tr
<xsl:output method="xml" omit-xml-declaration="yes" indent="yes" xalan:indent-amount="2"/>
<xsl:template match="/">
<Customer CustNumber="{/rdml:function/rdml:fields/rdml:field[@name='CUSTNUM']/@value}">
  <CustName><xsl:value-of select="/rdml:function/rdml:fields/rdml:field[@name='NAME']/@value"/></CustN
  <Street><xsl:value-of select="/rdml:function/rdml:fields/rdml:field[@name='STREET']/@value"/></Street>
  <City><xsl:value-of select="/rdml:function/rdml:fields/rdml:field[@name='CITY']/@value"/></City>
  <State><xsl:value-of select="/rdml:function/rdml:fields/rdml:field[@name='STATE']/@value"/></State>
  <PostCode><xsl:value-of select="/rdml:function/rdml:fields/rdml:field[@name='ZIP']/@value"/></PostCode
</Customer>
</xsl:template>
</xsl:transform>
```

Source Tab

The Source tab displays the XML source used. The XML Source is specified during the definition of a new project:

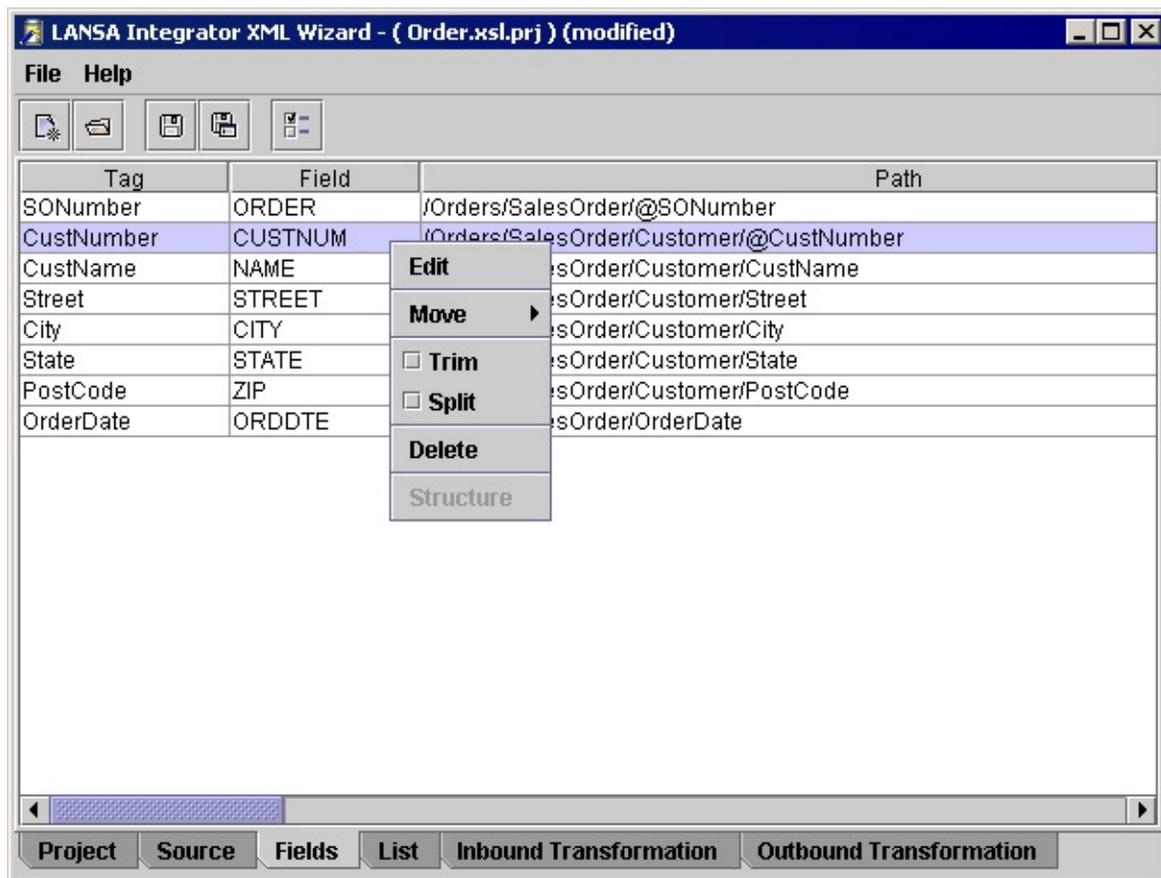


Fields Tab

This tab contains a table of elements and attributes which will be mapped to the RDML function fields when the Inbound/Outbound transformation style sheet is applied to the XML source.

You can use the [7.24 Repository Viewer](#) to drag and drop a LANSa field name onto a table row.

The entries in this table were added using the Add to fields or Add child tags to fields options in the [Project Tab](#):



Right mouse click on the selected row(s) to display the pop-up menu. You can then delete the selected row(s) or Edit the selected row.

If you select more than one row:

- Delete will remove them all from the collection.
- Edit will only alter the first row selected.

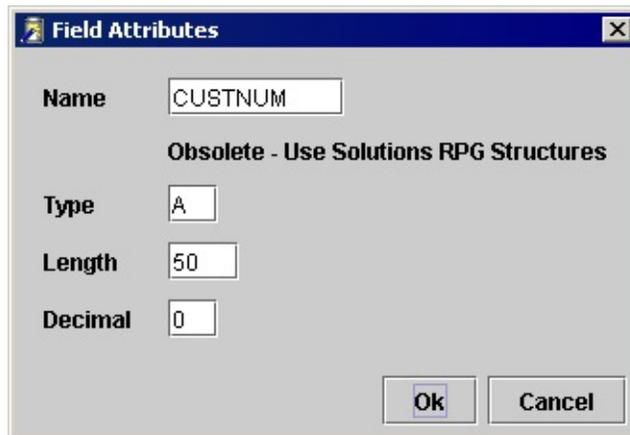
Edit

Allows you to change/edit the RDML field name to be mapped. The keyboard Enter key will perform the same function.



If the structure preference is enabled, extra fields for data type and length are made available.

It is recommended not to store 3GL information in the project. Use RPG Structures or the latest RPG service program that uses internal data type information and does not require external 3GL XML files.



Trim

Includes the normalize-space function on the inbound XML field tag.

```
<rdml:field name="NAME" value="{normalize-space(/Orders/SalesOrder/Customer/CustName)}"/>
```

Split

Includes the attribute split="yes" on the inbound XML field tag.

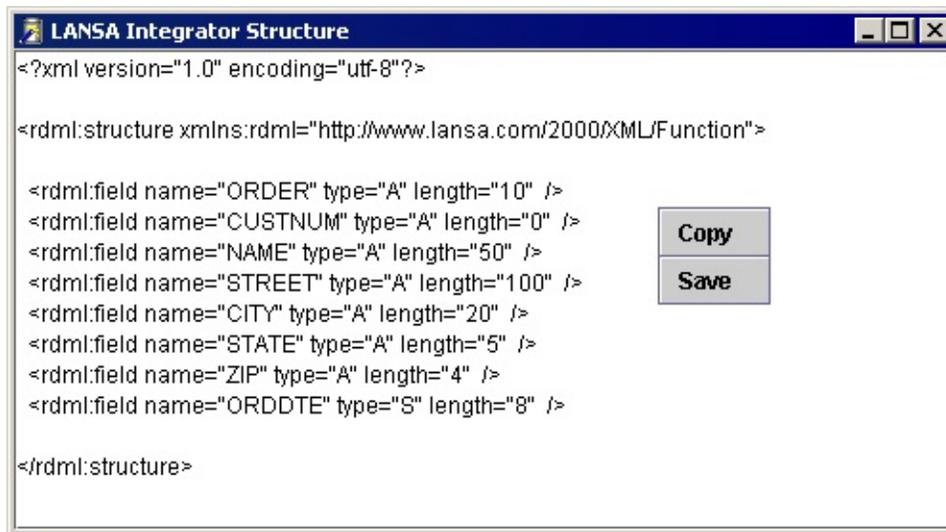
```
<rdml:field name="NAME" value="{/Orders/SalesOrder/Customer/CustName}" split="yes"/>
```

Delete

Allows you to remove the selected element(s) from the table. The keyboard Delete key will perform the same function.

Structure

Create the structure XML from the function fields. The sequence and data types of these fields must match the 3GL data structure.



```
<?xml version="1.0" encoding="utf-8"?>

<rdml:structure xmlns:rdml="http://www.lansa.com/2000/XML/Function">

  <rdml:field name="ORDER" type="A" length="10" />
  <rdml:field name="CUSTNUM" type="A" length="0" />
  <rdml:field name="NAME" type="A" length="50" />
  <rdml:field name="STREET" type="A" length="100" />
  <rdml:field name="CITY" type="A" length="20" />
  <rdml:field name="STATE" type="A" length="5" />
  <rdml:field name="ZIP" type="A" length="4" />
  <rdml:field name="ORDDTE" type="S" length="8" />

</rdml:structure>
```

Copy

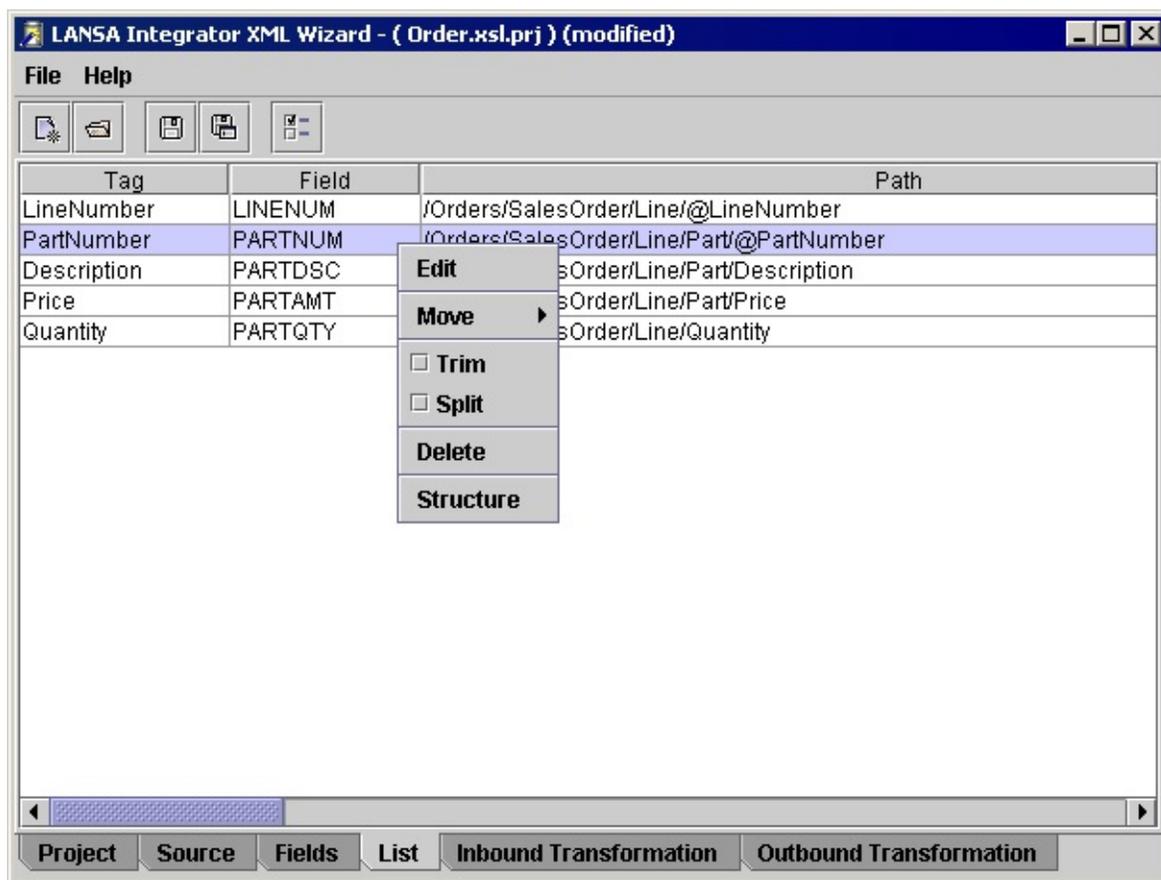
Save

List Tab

This tab contains a table of field elements and attributes which will be mapped to the RDML working list fields when the Inbound/Outbound transformation style sheet is applied to the XML source.

You can use the [7.24 Repository Viewer](#) to drag and drop a LANSa field name onto a table row.

The entries in this table were added using the List mark option in the [Project Tab](#):



Right mouse click on the selected node(s) to display the pop-up menu. The pop-up menu allows you to Delete the selected nodes or to Edit the selected Node's field.

If you select more than one node:

- Delete will remove all selected nodes from the collection.
- Edit will only alter the first node selected.

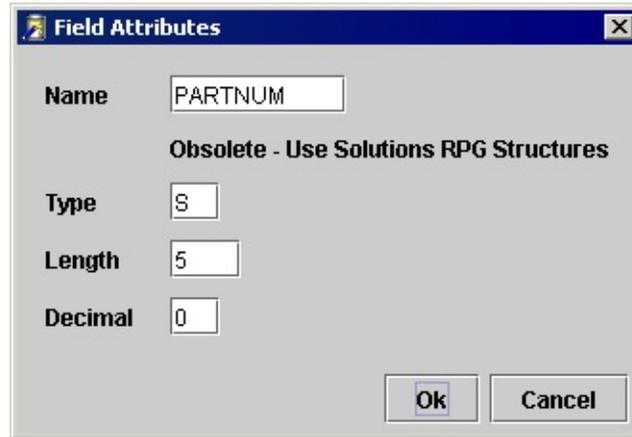
Edit

Allows you to change/edit the RDML working list field name to be mapped. The keyboard Enter key will perform the same function.



If the structure preference is enabled, extra fields for data Type and Length are made available.

It is recommended not to store 3GL information in the project. Use RPG Structures or the latest RPG service program that uses internal data type information and does not require external 3GL XML files.



Trim

Includes the normalize-space function on the inbound XML field tag.

```
<rdml:field name="NAME" value="{normalize-space(/Orders/SalesOrder/Customer/CustName)}"/>
```

Split

Includes the attribute split="yes" on the inbound XML field tag.

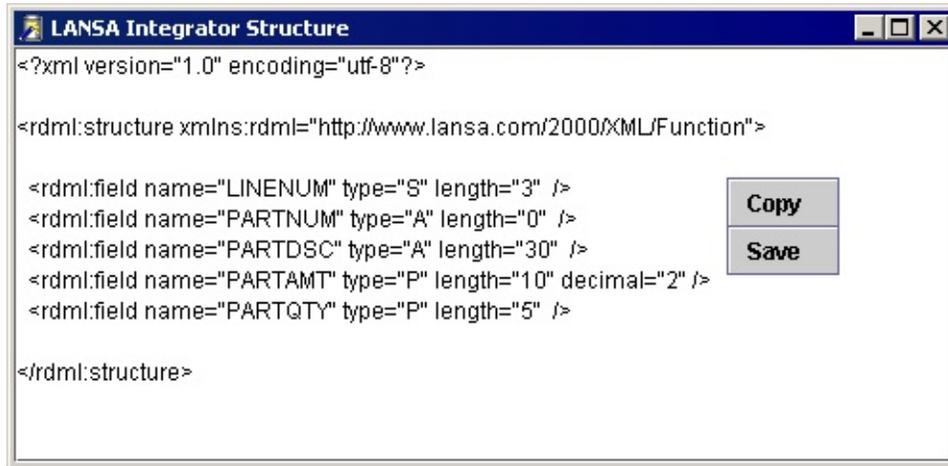
```
<rdml:field name="NAME" value="{/Orders/SalesOrder/Customer/CustName}" split="yes"/>
```

Delete

Allows you to remove the selected element(s) from the table. The keyboard Delete key will perform the same function.

Structure

Create the structure XML from the list fields. The sequence and data types of these fields must match the 3GL data structure:



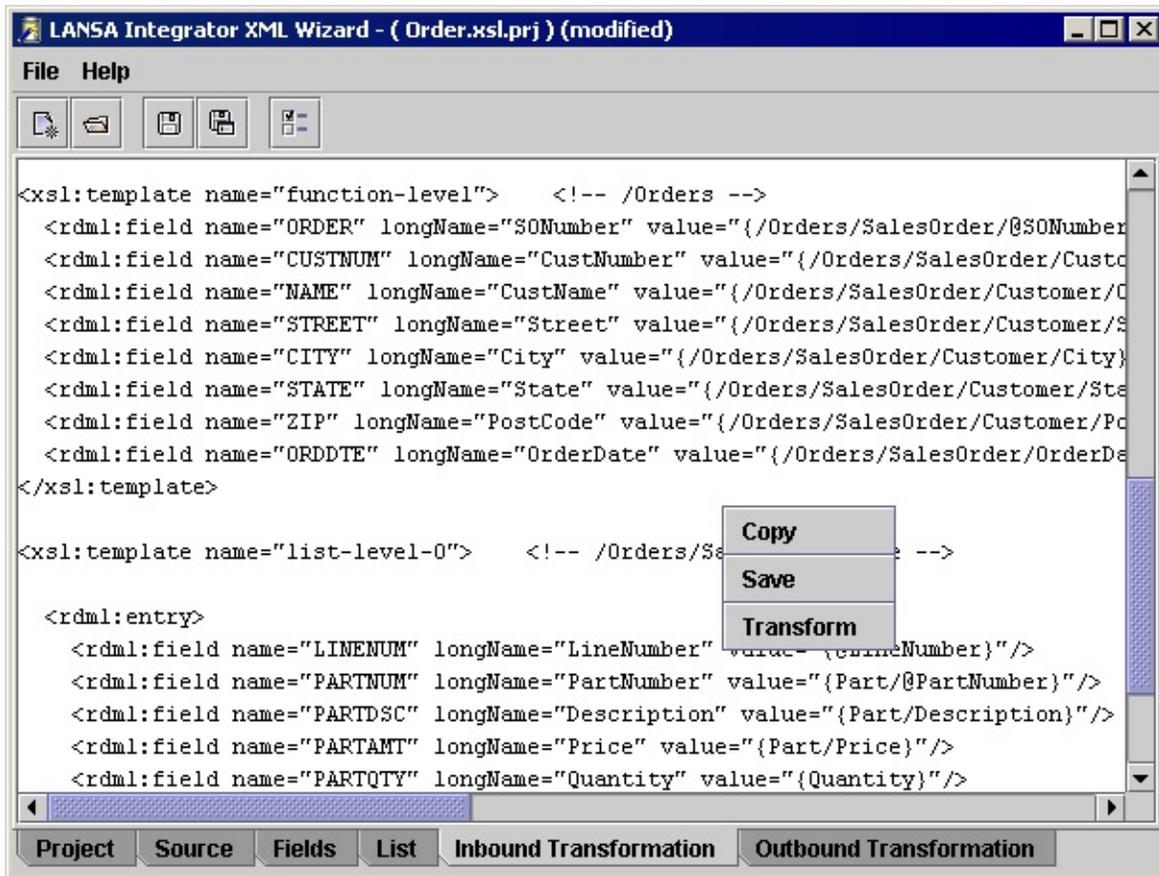
The screenshot shows a window titled "LANSA Integrator Structure" with a text area containing XML code. The code defines an rdm1:structure with five fields: LINENUM (type="S", length="3"), PARTNUM (type="A", length="0"), PARTDSC (type="A", length="30"), PARTAMT (type="P", length="10", decimal="2"), and PARTQTY (type="P", length="5"). To the right of the text area are two buttons: "Copy" and "Save".

```
<?xml version="1.0" encoding="utf-8"?>  
  
<rdml:structure xmlns:rdml="http://www.lansa.com/2000/XML/Function">  
  
  <rdml:field name="LINENUM" type="S" length="3" />  
  <rdml:field name="PARTNUM" type="A" length="0" />  
  <rdml:field name="PARTDSC" type="A" length="30" />  
  <rdml:field name="PARTAMT" type="P" length="10" decimal="2" />  
  <rdml:field name="PARTQTY" type="P" length="5" />  
  
</rdml:structure>
```

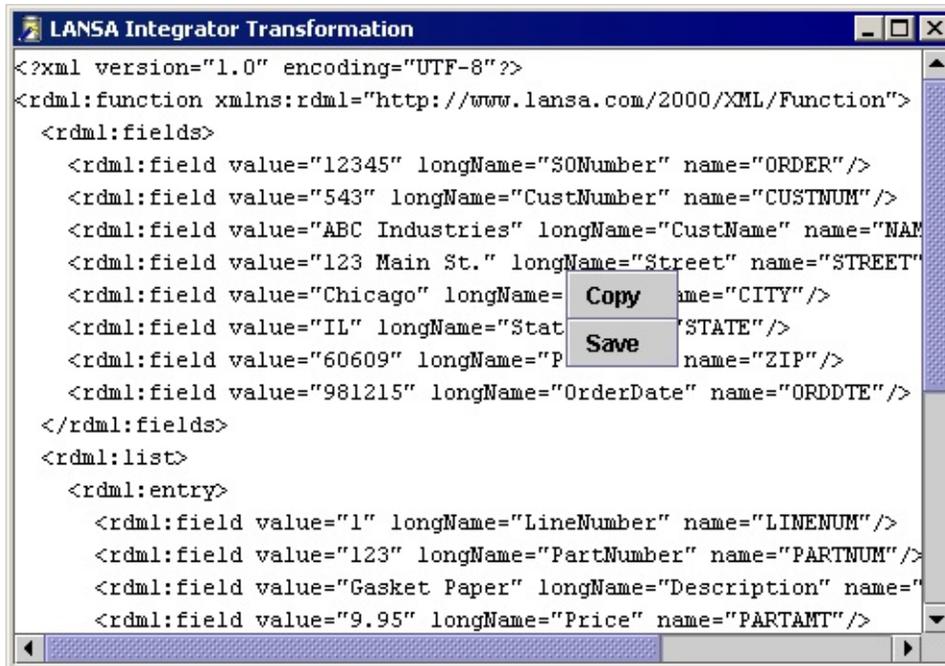
Inbound Transformation Tab

This transformation style sheet will be applied to the incoming (inbound) XML and will perform mapping from the received XML document to the RDML function fields and one working list.

The contents of this tab cannot be directly altered. The XLS is automatically generated as the fields and list are marked.



This screen capture is of an inbound transformation test:



LANSA Integrator Transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<rdml:function xmlns:rdml="http://www.lansa.com/2000/XML/Function">
  <rdml:fields>
    <rdml:field value="12345" longName="SONumber" name="ORDER" />
    <rdml:field value="543" longName="CustNumber" name="CUSTNUM" />
    <rdml:field value="ABC Industries" longName="CustName" name="NAM
    <rdml:field value="123 Main St." longName="Street" name="STREET"
    <rdml:field value="Chicago" longName="City" name="CITY" />
    <rdml:field value="IL" longName="State" name="STATE" />
    <rdml:field value="60609" longName="PostalCode" name="ZIP" />
    <rdml:field value="981215" longName="OrderDate" name="ORDDTE" />
  </rdml:fields>
  <rdml:list>
    <rdml:entry>
      <rdml:field value="1" longName="LineNumber" name="LINENUM" />
      <rdml:field value="123" longName="PartNumber" name="PARTNUM" />
      <rdml:field value="Gasket Paper" longName="Description" name="
      <rdml:field value="9.95" longName="Price" name="PARTAMT" />
    </rdml:entry>
  </rdml:list>
</rdml:function>
```

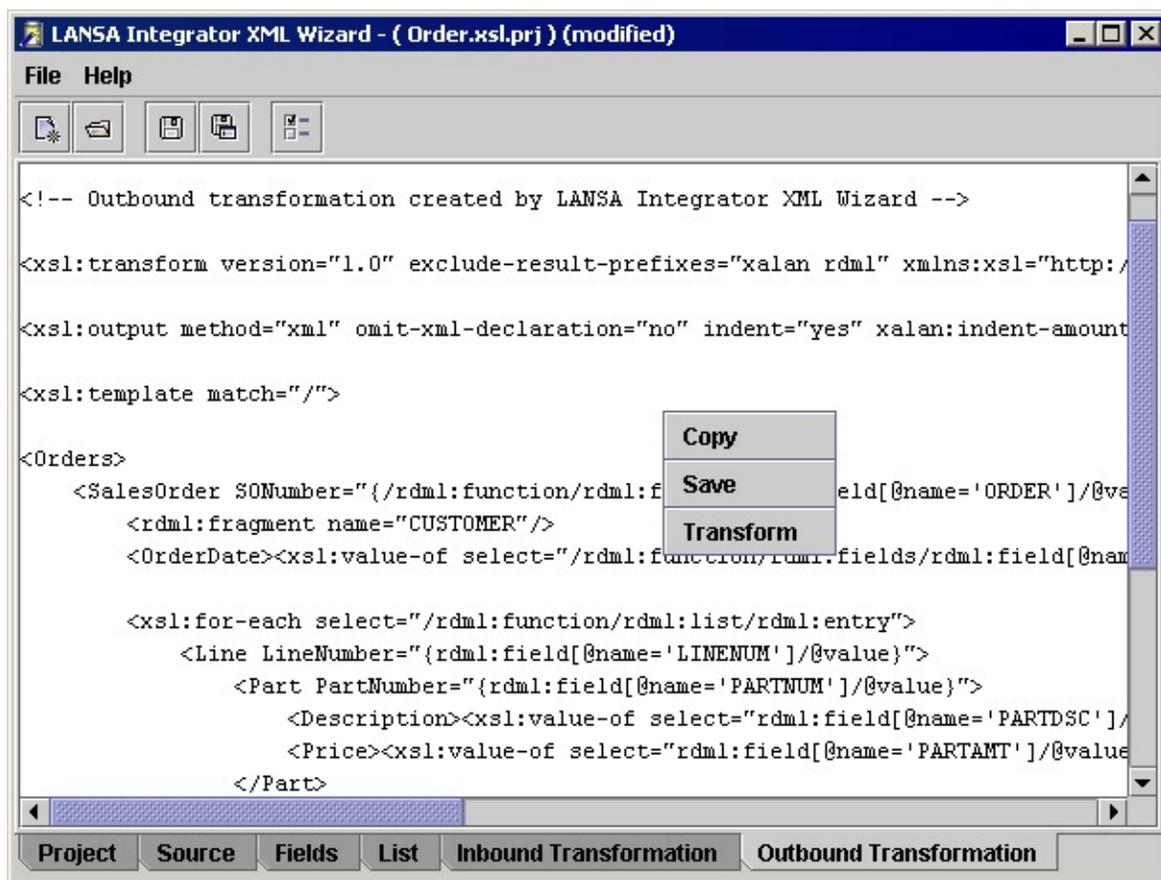
Copy
Save

Outbound Transformation Tab

This transformation style sheet will be applied to the RDML function fields and working list to create the outgoing (outbound) XML document. Fragments allow complex XML documents to be built up by multiple transformation calls.

Note: The usage of fragments is only possible if the JSM Service supports this capability, i.e. the HTTPService, XMLFileService and XMLQueueService.

The contents of this tab cannot be directly altered. The XLS is automatically generated as the fields and list are marked.



This screen capture is of an outbound transformation test:

LANSA Integrator Transformation

```
<?xml version="1.0" encoding="UTF-8"?>
<Orders>
  <SalesOrder SONumber="12345">
    <rdml:fragment xmlns:rdml="http://www.lansa.com/2000/XML/Function"
    <OrderDate>981215</OrderDate>
    <Line LineNumber="1">
      <Part PartNumber="123">
        <Description>Gasket Paper</Description>
        <Price>9.95</Price>
      </Part>
      <Quantity>10</Quantity>
    </Line>
    <Line LineNumber="2">
      <Part PartNumber="456">
        <Description>Gasket <Polymer> Glue</Description>
        <Price>13.27</Price>
      </Part>
      <Quantity>5</Quantity>
    </Line>
  </SalesOrder>
</Orders>
```

Copy

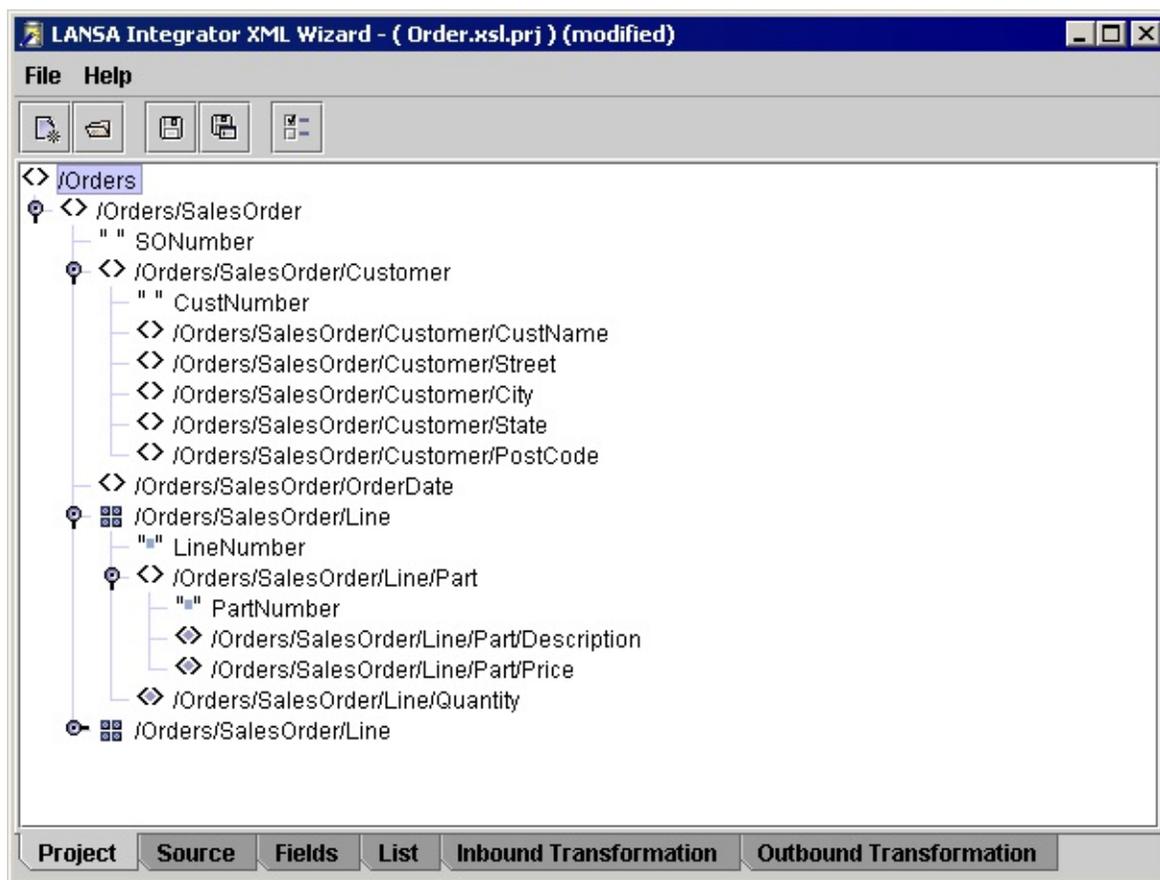
Save

Map Fields

Once you have opened the project, the XML Transformation Wizard will allow you to map the XML data to your LANSAs fields or lists.

The sample XML document can be viewed using the [Source Tab](#). This tab is for display purposes only.

The [Project Tab](#) will show the structure of the XML document. Using the pop-up menu, you may mark the XML data and map it to LANSAs fields or to LANSAs fields in lists.



To specify the field names for the marked XML tags, you will use the [Fields Tab](#) or the [List Tab](#).

You can use the [7.24 Repository Viewer](#) to drag and drop a LANSAs field name onto a table row.

LANSA Integrator XML Wizard - (Order.xml.prj) (modified)

File Help

Tag	Field	Path
SONumber	ORDER	/Orders/SalesOrder/@SONumber
CustNumber	CUSTNUM	/Orders/SalesOrder/Customer/@CustNumber
CustName	NAME	/Orders/SalesOrder/Customer/CustName
Street	STREET	/Orders/SalesOrder/Customer/Street
City	CITY	/Orders/SalesOrder/Customer/City
State	STATE	/Orders/SalesOrder/Customer/State
PostCode	ZIP	/Orders/SalesOrder/Customer/PostCode
OrderDate	ORDDTE	/Orders/SalesOrder/OrderDate

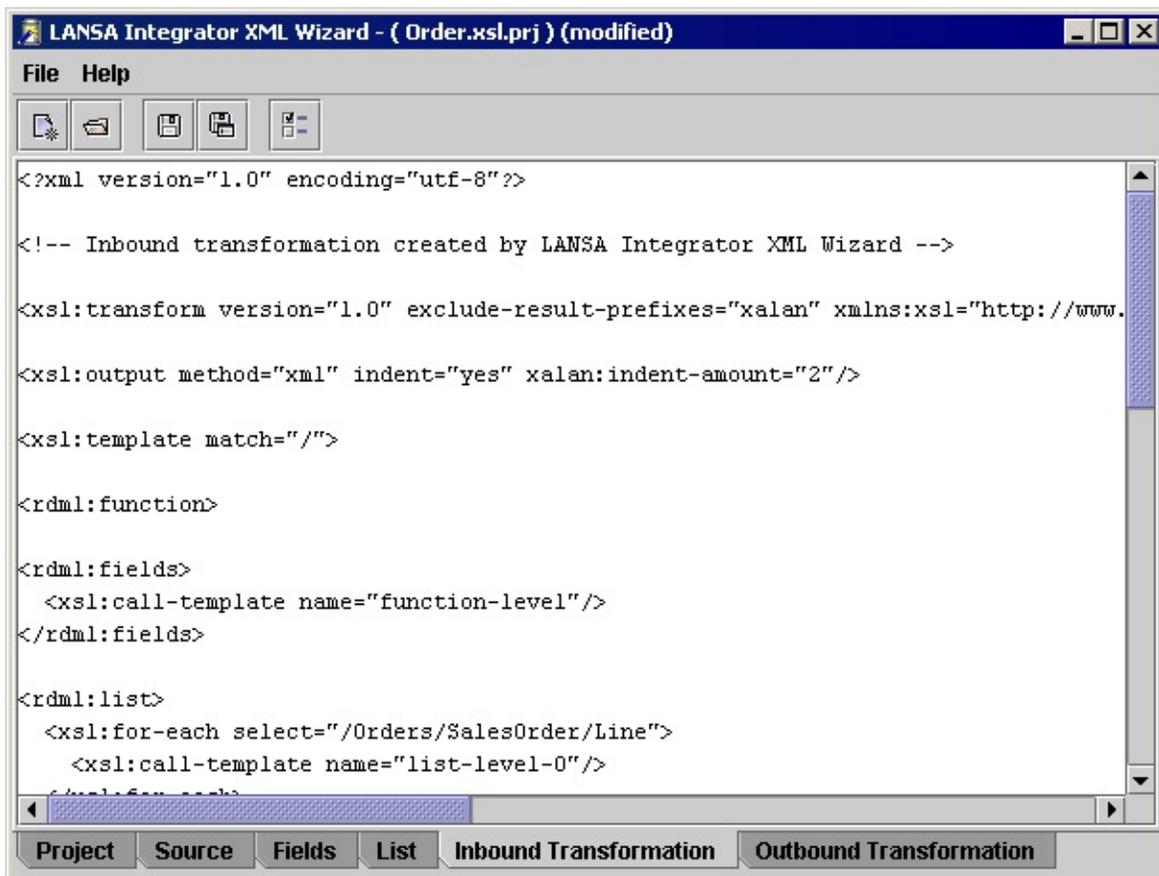
Project Source Fields List Inbound Transformation Outbound Transformation

Once you have identified the fields or lists, you are ready to test and save your transformations using the [Inbound Transformation Tab](#) and [Outbound Transformation Tab](#).

Test Transformation

Inbound and outbound style sheets can be tested by selecting the Transform menu item from the pop-up menu available on each of the tab panels.

When testing the outbound transformation, you will be prompted to enter an XML file containing a sample of FunctionXML.



Save Transformation

Once you have tested your transformations, you must save the XSL style sheets. After you have saved the XSL files, you must copy these files to the server, so they can be accessed by a service.

By default, the style sheets are copied to the /jsm/instance/xsl directory.

To make a style sheet available to a service an entry needs to be added to the service's property resource.

The resource file associated with a service can be determined by looking for the resource entry in the service.properties file in the system sub directory.

Using the resource name locate the service resource properties file in the properties sub directory.

Add an entry to the service resource properties file with following format:

```
xsl.name=xsl/filename.xsl
```

where:

name is the identifier used in the command XSL keyword.

filename is the name of the XSL file which was saved by the XML Transformation Wizard and copied to the server.

Example

```
SERVICE_LOAD SERVICE(HTTPService)
```

```
RECEIVE XSL(RECEIVEORDER)
```

```
SEND XSL(SENDORDER)
```

From the service.properties file, the HTTPService service is using the HTTPService.properties for resources.

Locate the HTTPService.properties file in the properties sub directory.

Add the following entries:

```
xsl.sendorder=xsl/send-order.xsl
```

```
xsl.receiveorder=xsl/receive-order.xsl
```


7.24 Repository Viewer

The Repository Viewer is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The Repository Viewer is an application that allows you to view LANSAs repository information. The Repository Viewer uses JSMDirect, LANSAs functions and the RFIDataSourceService to select repository information.

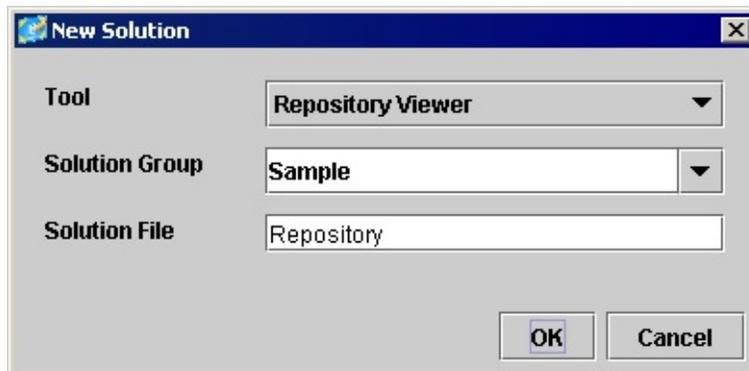
Use the Repository Viewer to drag and drop LANSAs field names onto Wizard mappings that require LANSAs fields.

Review the following topics:

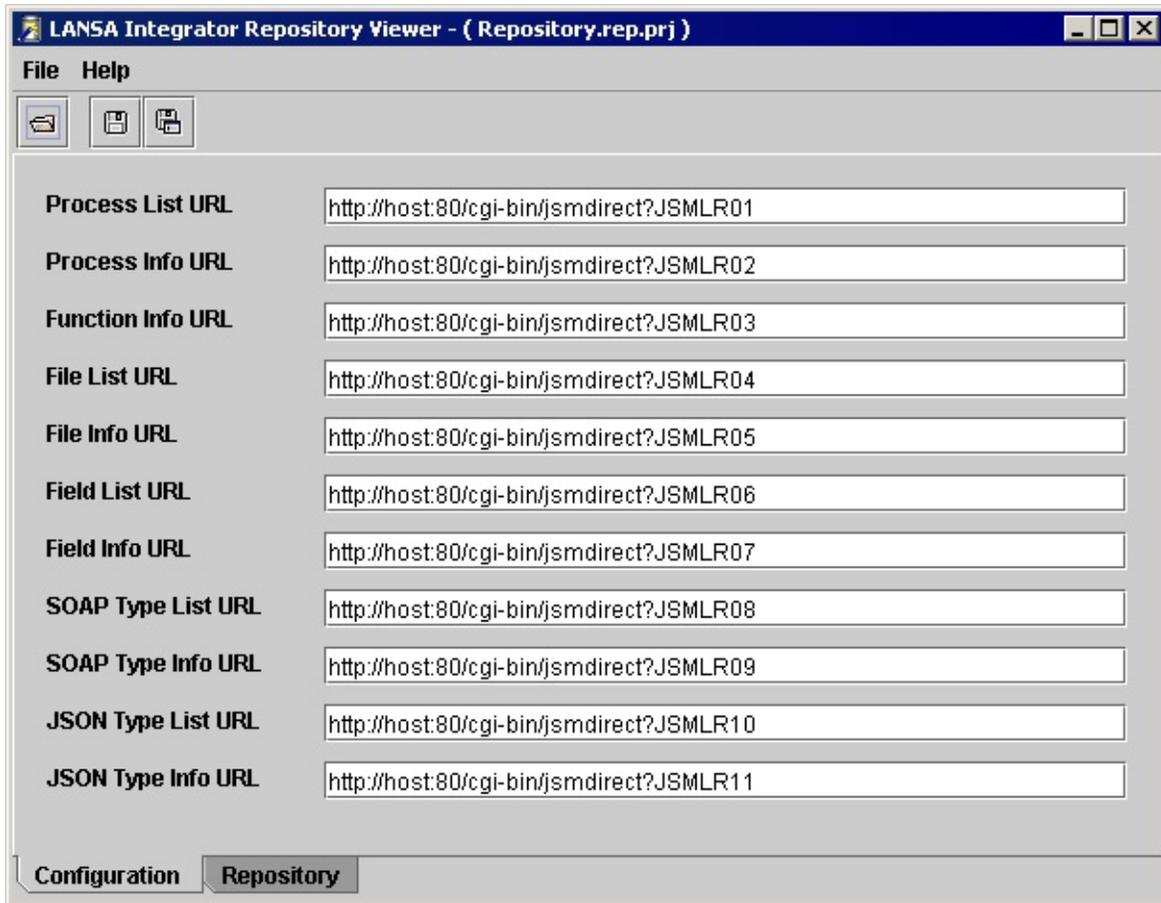
- [7.24.1 Create a Repository Project](#)
- [7.24.2 LANSAs Repository Functions](#)

7.24.1 Create a Repository Project

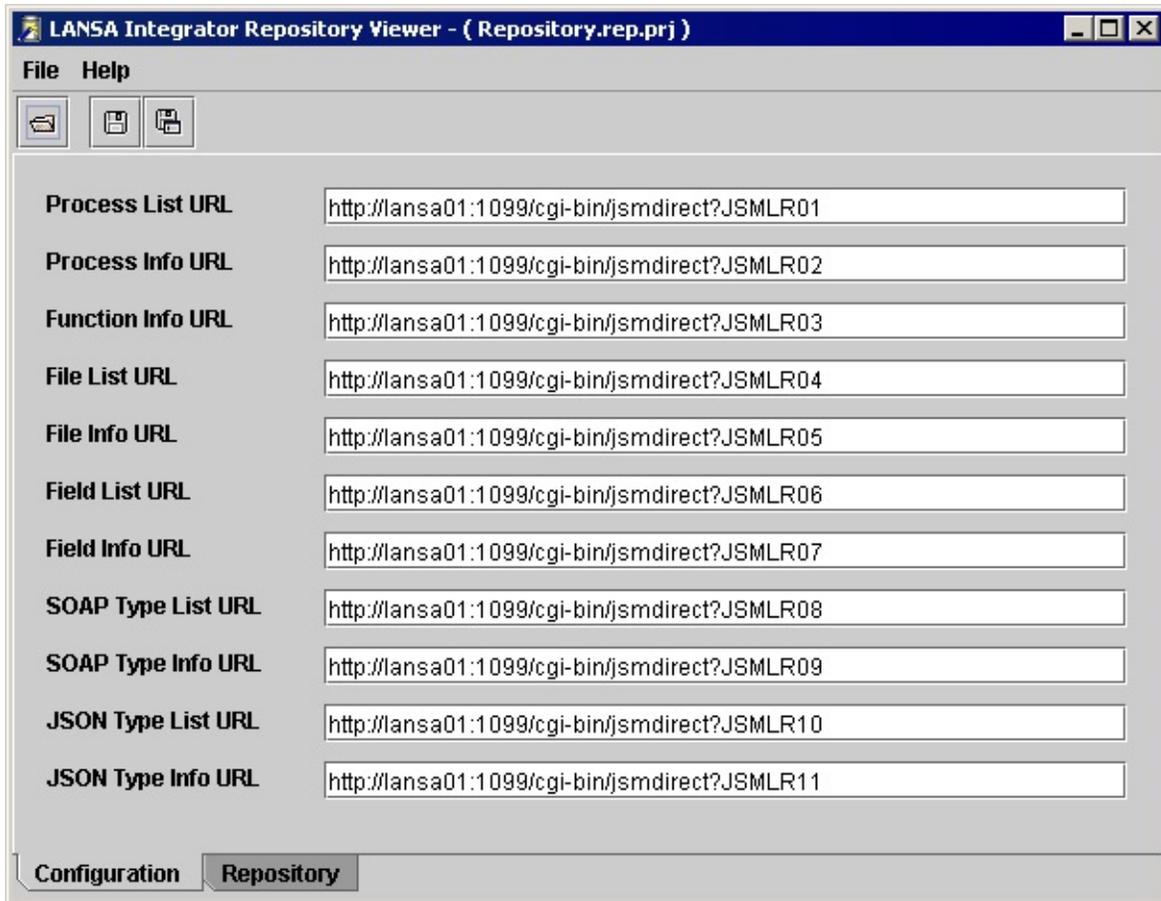
1. In Integrator Studio, from the Project Solutions node right click on the *New Solution* menu item.
2. Select the *Repository Viewer* tool from the context menu and enter or select the *Group* folder to receive the project file.
3. Enter the project File name. The extension '.rep.prj' is added automatically.



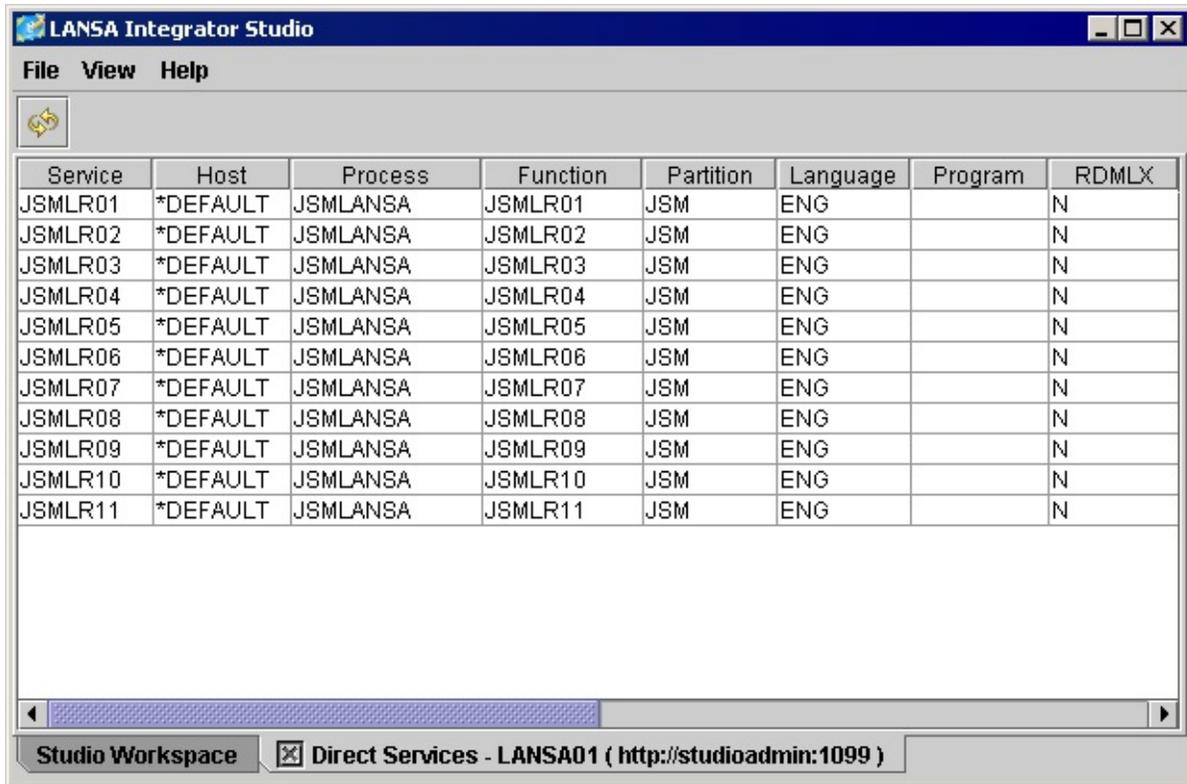
4. Press the *OK* button to create the project, to open the LANSa Integrator Repository Viewer.



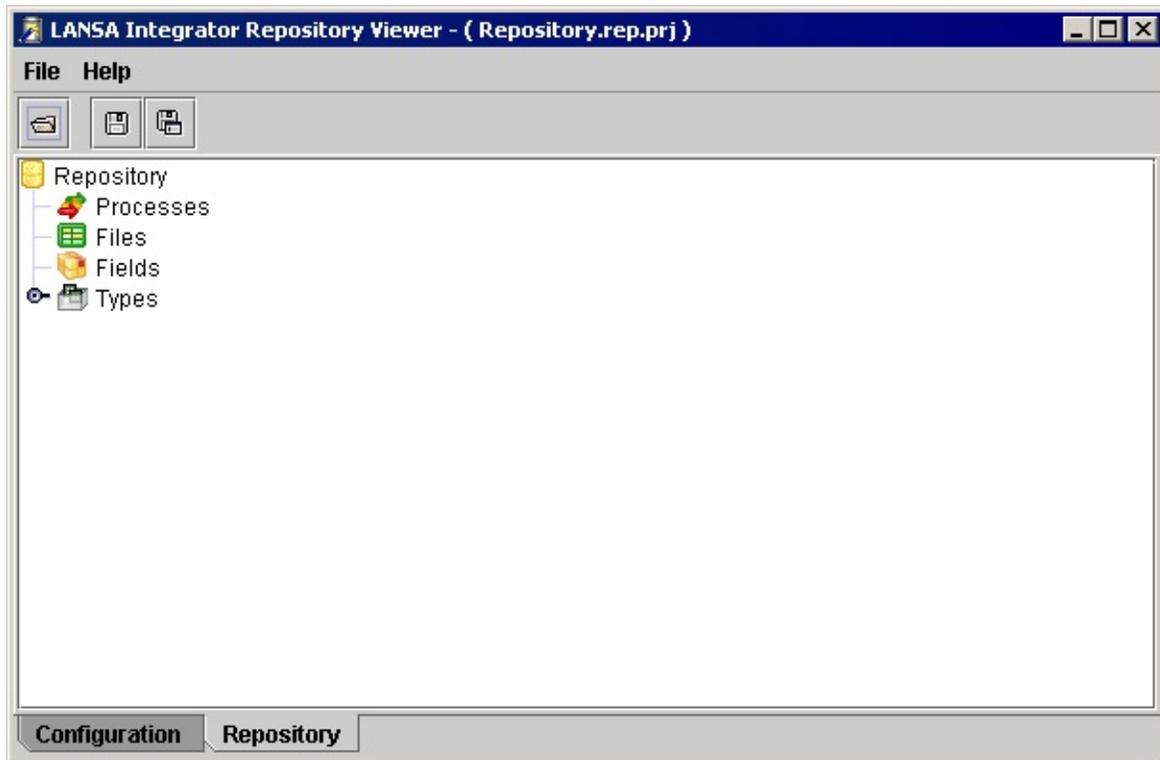
5. By default each of the seven information programs have a sample URL. Change these sample URLs to suit your own environment.



6. Create Direct Service entries to support the Repository Viewer URL service requests. These entries specify the LANSA functions that will be executed and the partition that will be used.

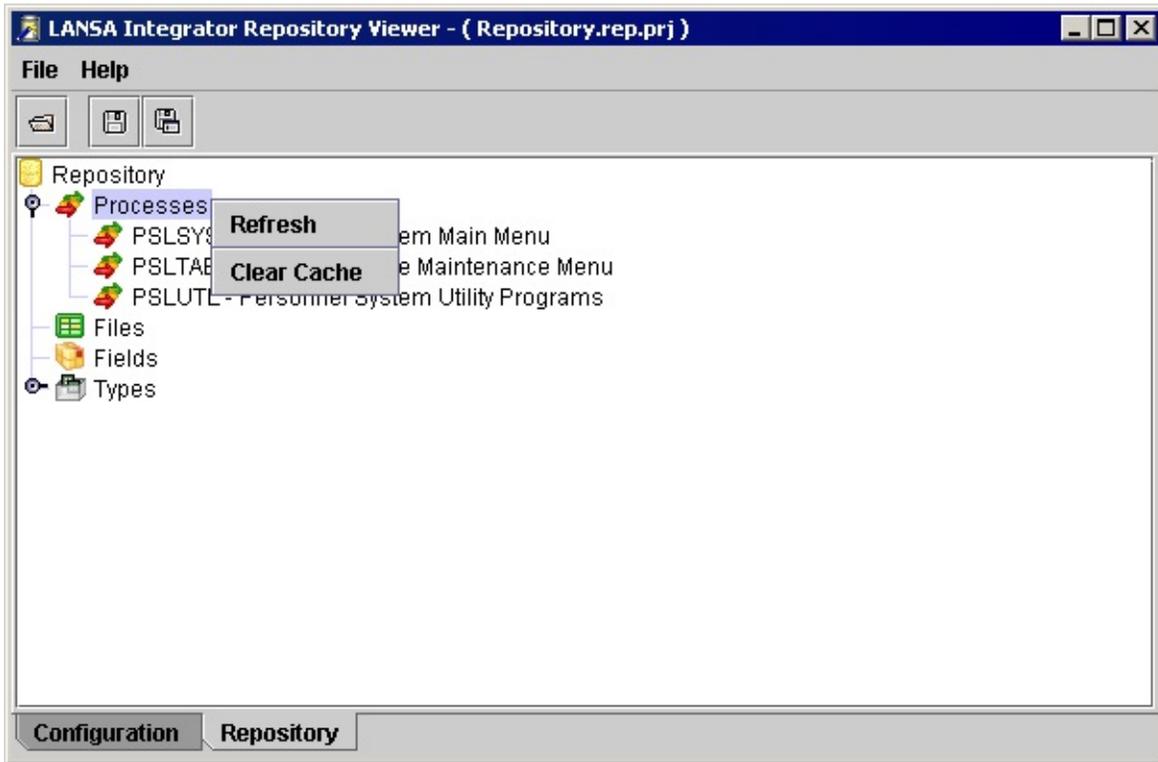


7. View the LANSA repository by selecting the Repository tab.

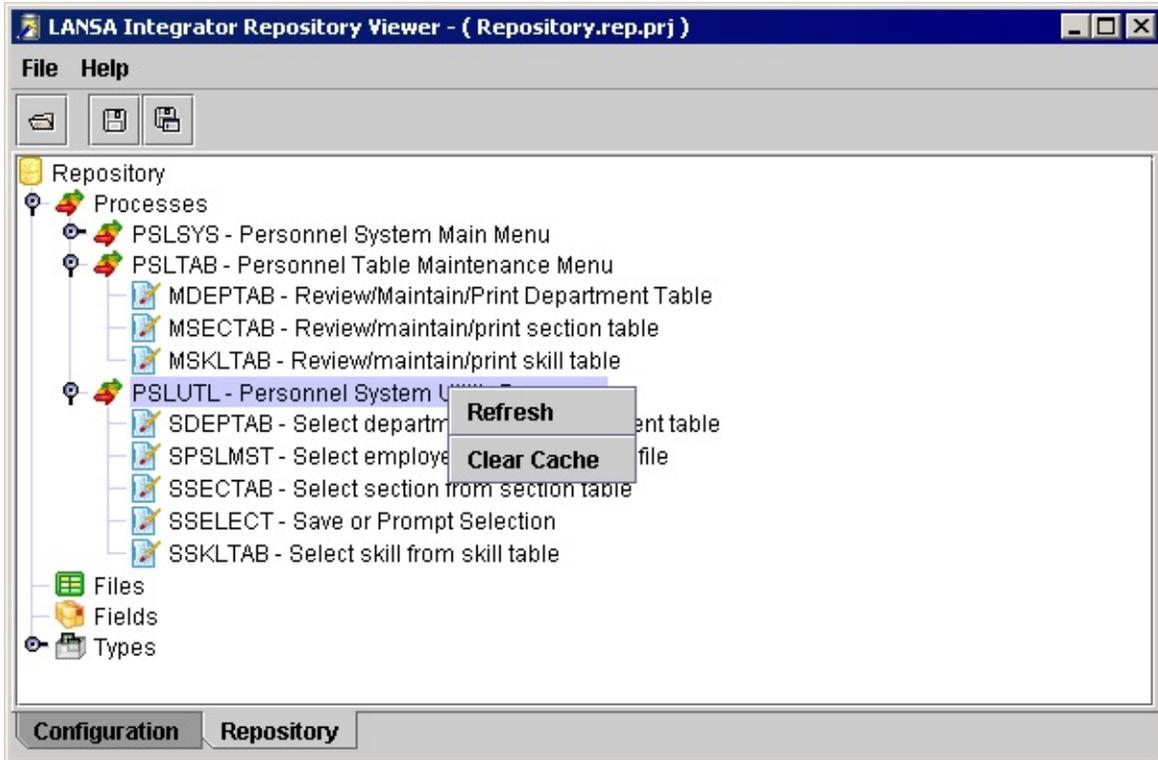


8. To view processes, double-click the Processes node or select the *Refresh*

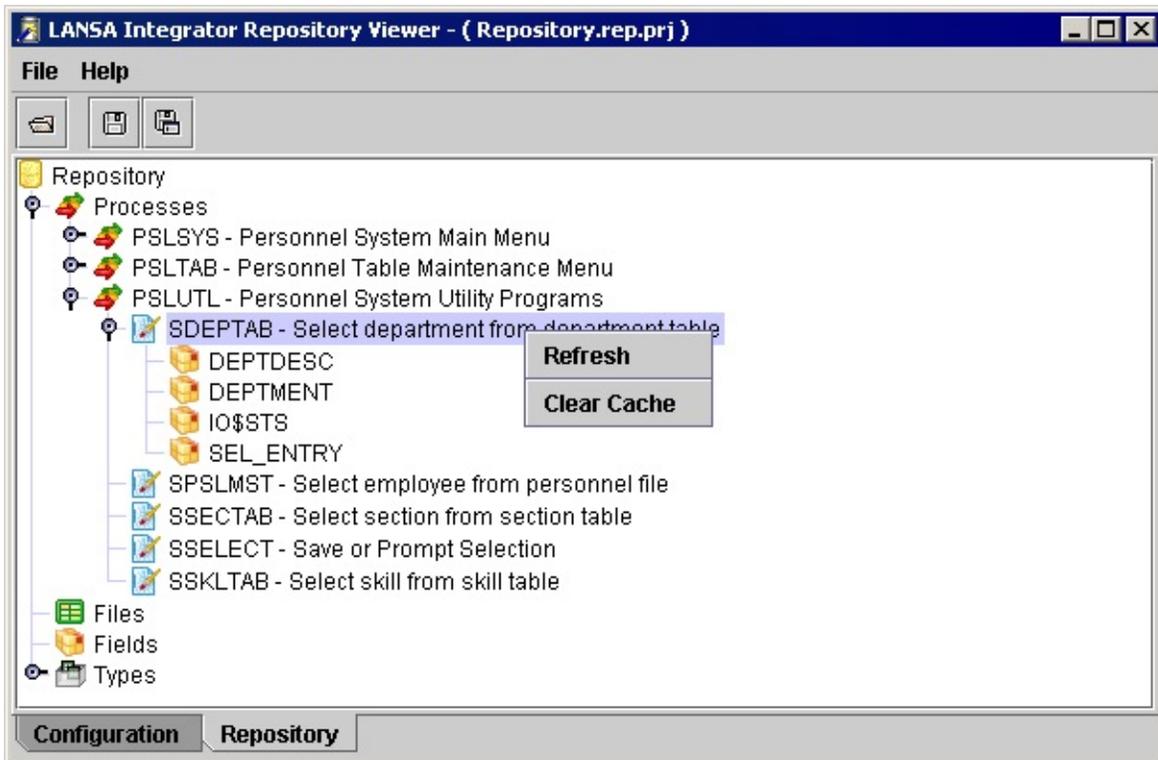
menu item.



9. To view functions used in a particular process, double-click the process node or select the *Refresh* menu item.

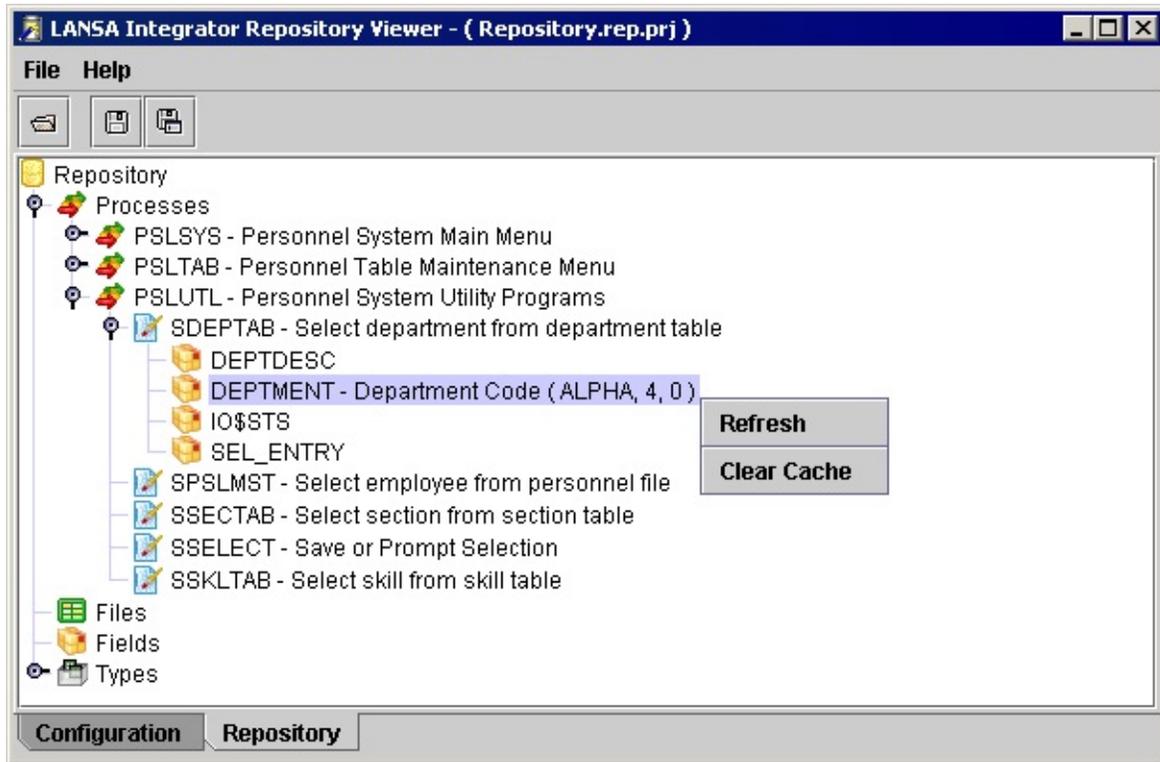


10. To view fields that have been used in a particular function, double-click the function node or select the *Refresh* menu item.

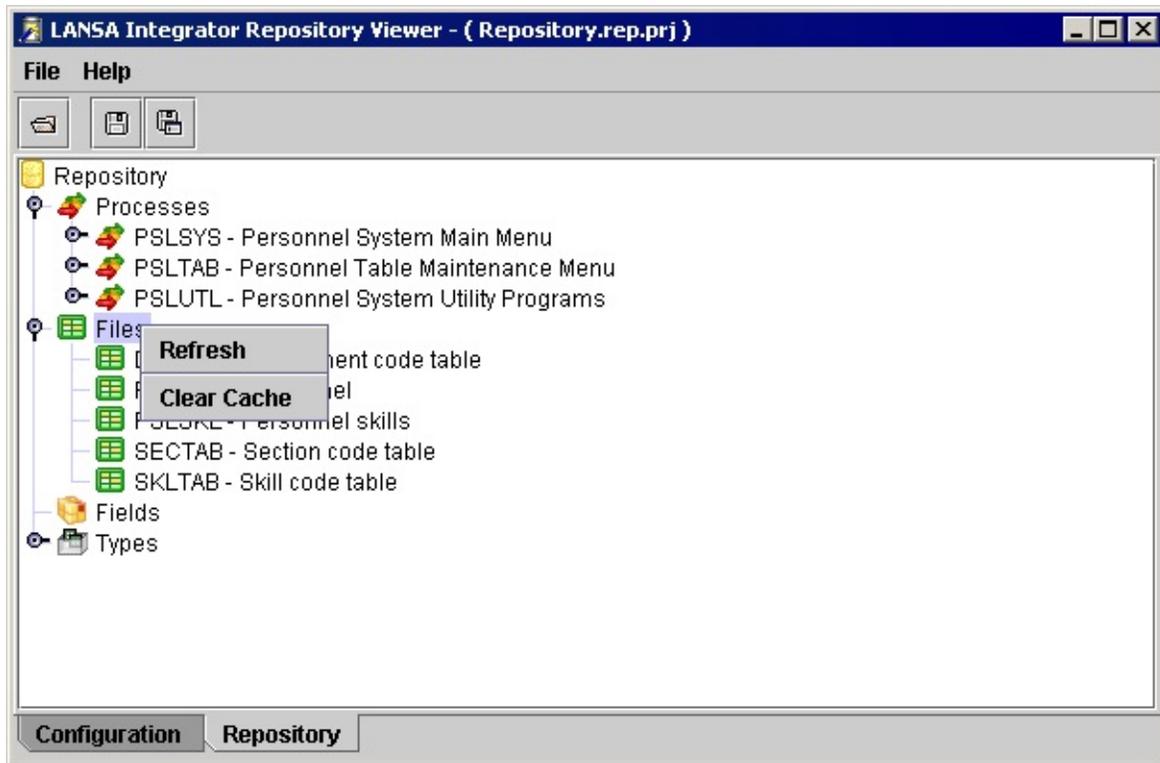


11. To view information on a particular field, double-click the field node or select the *Refresh* menu item.

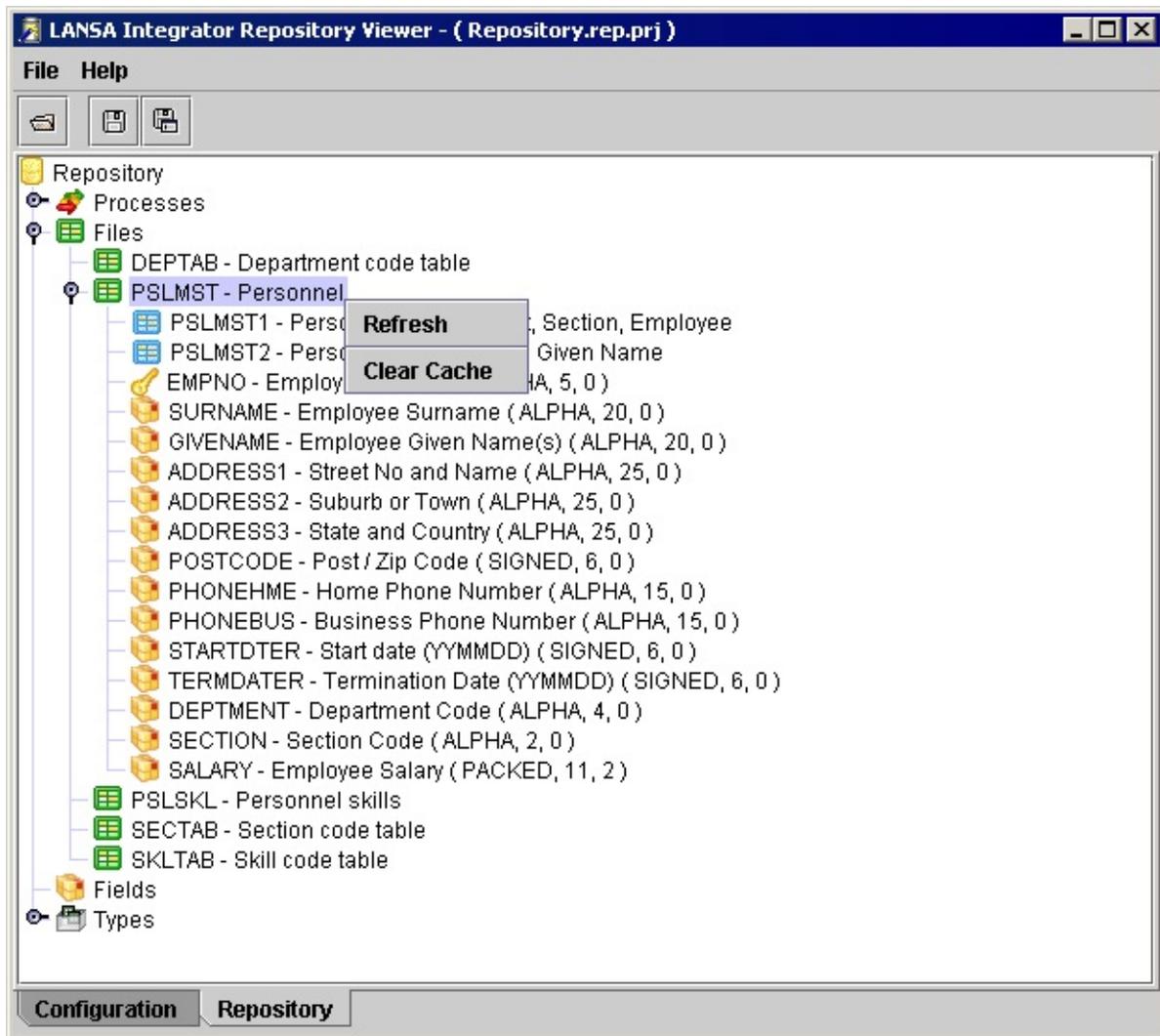
When field information is selected, it is cached and used in all areas of the repository viewer where field information is displayed.



To view files, double-click the Files node or select the *Refresh* menu item.

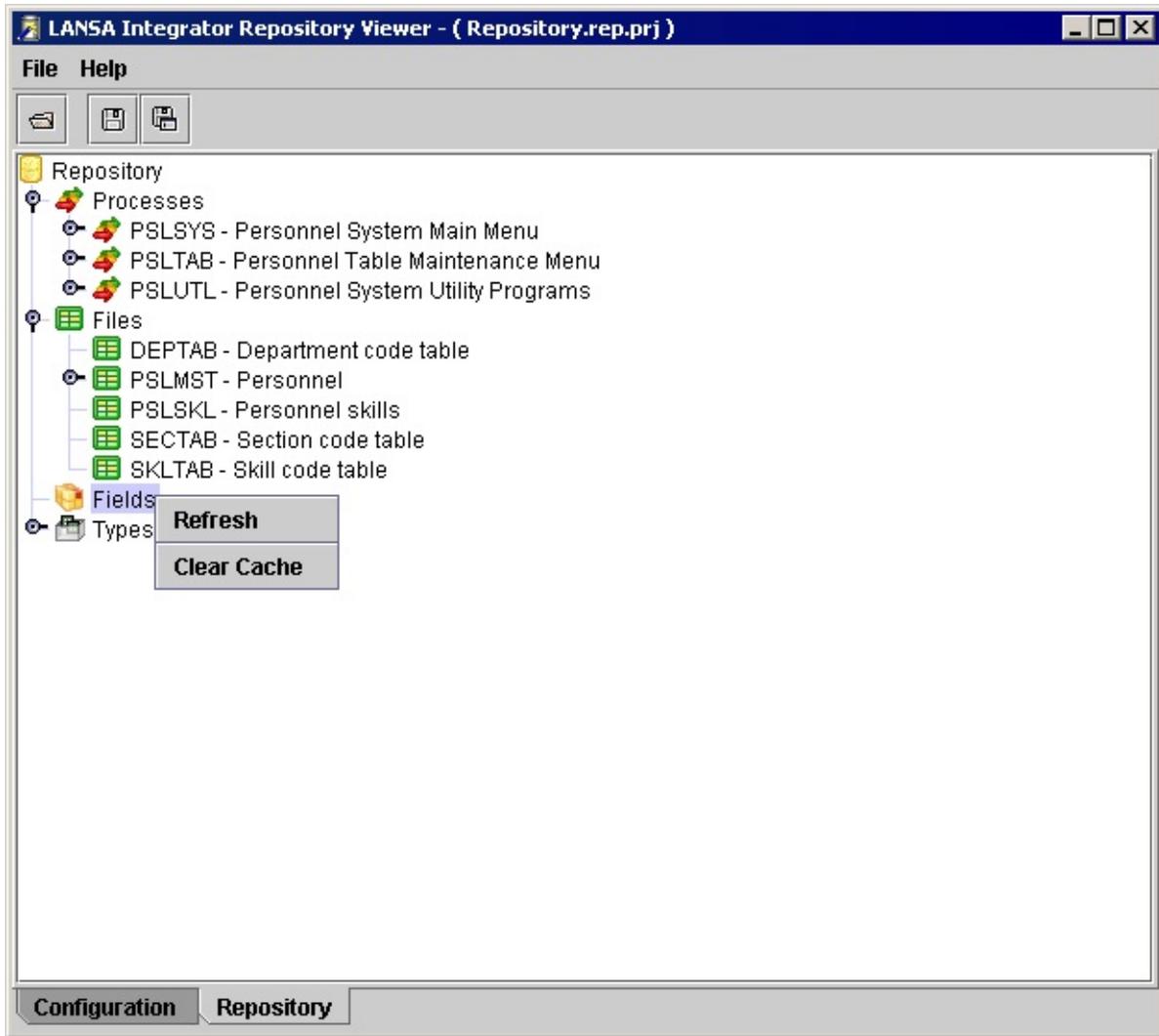


12. To view fields that have been used in a particular file, double-click the file node or select the *Refresh* menu item. The file information process also selects repository field information on each field in the file.

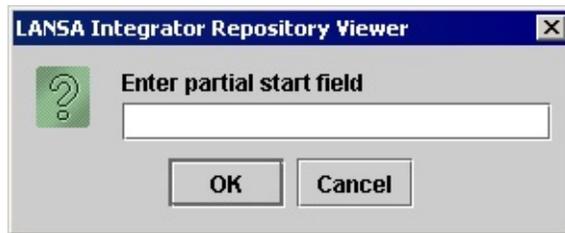


When field information is selected, it is cached and used in all areas of the repository viewer where field information is displayed.

13. To view repository field information, double-click the Fields node or select the *Refresh* menu item.



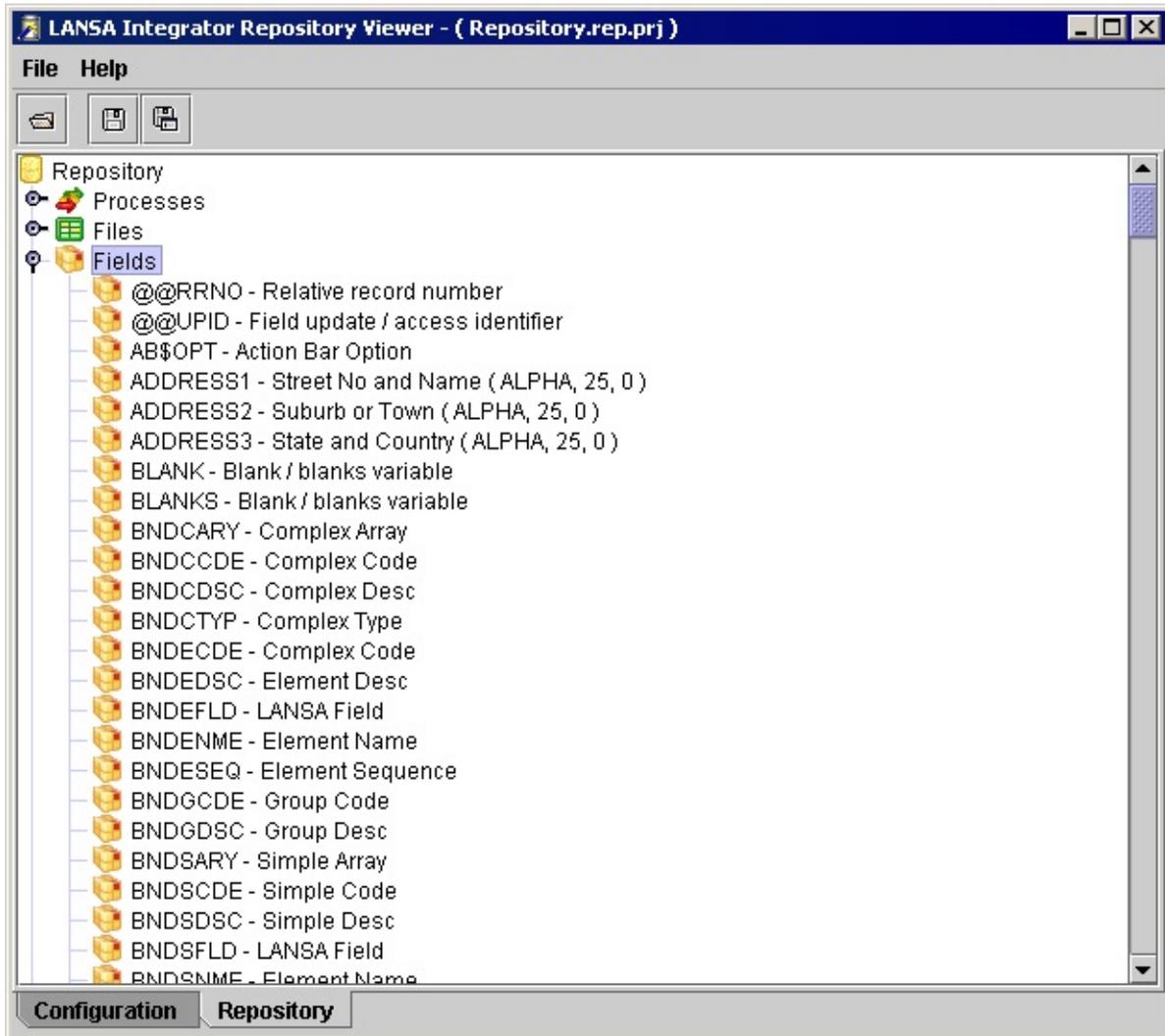
When you select *Refresh* from the context menu, you are offered a partial search, because there can be a large number of fields in a repository.



If the number of fields in your repository is less than 10,000 or if you want the first 9999 fields returned, then leave the value blank.

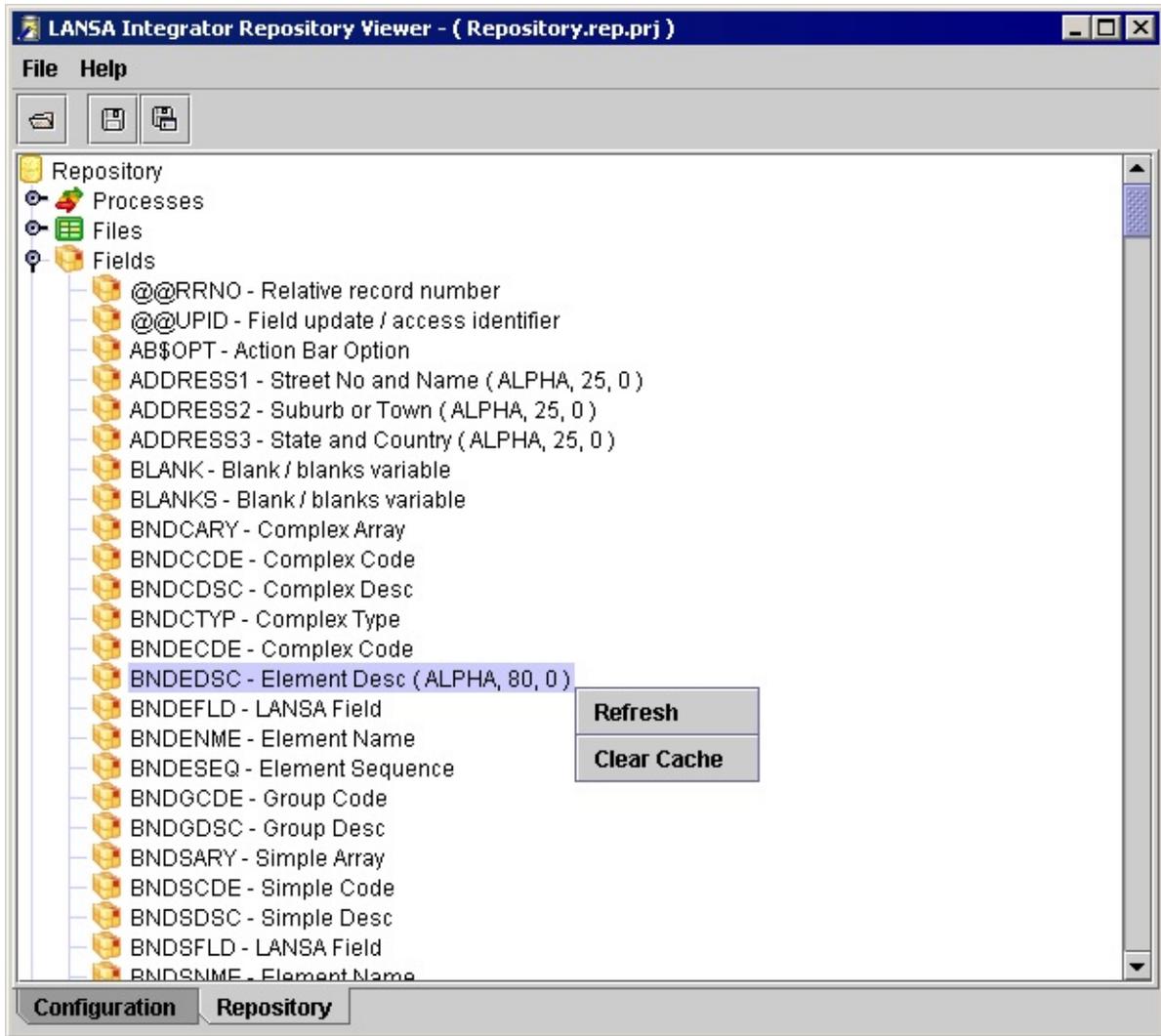
In the following screen capture you can see that all fields in the repository have been selected. No field information is available except for fields that have been cached from previous field information selections (in this example,

see example, ADDRESS1, ADDRESS2 and ADDRESS3 fields).



To view field information for a particular field, double-click the field node or select the *Refresh* menu item.

When field information is selected, it is cached and used in all areas of the repository viewer where field information is displayed.



7.24.2 LANSAs Repository Functions

Create a process to hold the following functions used by the repository viewer.

- [JSMLR01 LANSAs Process List](#)
- [JSMLR02 LANSAs Process Information](#)
- [JSMLR03 LANSAs Function Information](#)
- [JSMLR04 LANSAs File List](#)
- [JSMLR05 LANSAs File Information](#)
- [JSMLR06 LANSAs Field List](#)
- [JSMLR07 LANSAs Field Information](#)
- [JSMLR08 LANSAs SOAP Type List](#)
- [JSMLR09 LANSAs SOAP Type Information](#)
- [JSMLR10 LANSAs JSON Type List](#)
- [JSMLR11 LANSAs JSON Type Information](#)
- [JSMLR0A LANSAs Type List](#)
- [JSMLR0B LANSAs Type Information](#)

The following program example illustrates the JSMDirect service entries for these functions.

The partition and language values need to be changed to suit your environment.

```
JSMLR01 *DEFAULT JSMLANSA JSMLR01 JSM ENG
JSMLR02 *DEFAULT JSMLANSA JSMLR02 JSM ENG
JSMLR03 *DEFAULT JSMLANSA JSMLR03 JSM ENG
JSMLR04 *DEFAULT JSMLANSA JSMLR04 JSM ENG
JSMLR05 *DEFAULT JSMLANSA JSMLR05 JSM ENG
JSMLR06 *DEFAULT JSMLANSA JSMLR06 JSM ENG
JSMLR07 *DEFAULT JSMLANSA JSMLR07 JSM ENG
JSMLR08 *DEFAULT JSMLANSA JSMLR08 JSM ENG
JSMLR09 *DEFAULT JSMLANSA JSMLR09 JSM ENG
JSMLR10 *DEFAULT JSMLANSA JSMLR10 JSM ENG
JSMLR11 *DEFAULT JSMLANSA JSMLR11 JSM ENG
```

The following program example shows the LANSAs repository defined fields for the Studio Type database files and functions.

BNDCARY	Complex Array	A	1
BNDCCDE	Complex Code	A	5
BNDCDSC	Complex Desc	A	80
BNDCETY	Complex Entity	A	40
BNDCTYP	Complex Type	A	50
BNDECDE	Complex Code	A	5
BNDEDSC	Element Desc	A	80
BNDEETY	Element Entity	A	40
BNDENME	Element Name	A	40
BNDESEQ	Element Sequence	P	3 0
BNDGCDE	Group Code	A	5
BNDGDSC	Group Desc	A	80
BNDSARY	Simple Array	A	1
BNDSCDE	Simple Code	A	5
BNDSASC	Simple Desc	A	80
BNDSETY	Element Entity	A	40
BNDSNME	Element Name	A	40
BNDSTYP	Simple Type	A	20

The following program example shows the Studio Type database files.

BNDCSTPS Bind Complex Types

10	BNDGCDE	Group Code	1
20	BNDCCDE	Complex Code	2
30	BNDCDSC	Complex Desc	
40	BNDCTYP	Complex Type	
50	BNDCETY	Complex Entity	

BNDESTPS Binding Complex Type Elements

10	BNDGCDE	Group Code	1
20	BNDECDE	Complex Code	2
30	BNDESEQ	Element Sequence	3
40	BNDEDSC	Element Desc	

50	BNDSCDE	Simple Code
60	BNDCCDE	Complex Code
70	BNDCARY	Complex Array
80	BNDENME	Element Name
90	BNDEETY	Element Entity

BNDGTPS Binding Group Types

10	BNDGCDE	Group Code	1
20	BNDGDSC	Group Desc	

BNDSTPS Binding Simple Types

10	BNDGCDE	Group Code	1
20	BNDSCDE	Simple Code	2
30	BNDSDSC	Simple Desc	
40	BNDSTYP	Simple Type	
50	BNDSARY	Simple Array	
60	BNDSNME	Element Name	
70	BNDSETY	Element Entity	

JSMLR01 LANSА Process List

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#BNDMAPS) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDMAPL) TYPE(*CHAR) LENGTH(30)
DEFINE FIELD(#BNDPROSTR) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDPRONME) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDPRODSC) TYPE(*CHAR) LENGTH(40)
DEFINE FIELD(#BNDPROFIL) TYPE(*CHAR) LENGTH(10)

DEF_LIST NAME(#MAPLST) FIELDS((#BNDMAPS) (#BNDMAPL)) TY

DEF_LIST NAME(#PROLST) FIELDS((#BNDPRONME) (#BNDPRODSC

DEF_LIST NAME(#PROTABLE) FIELDS((#BNDPRONME) (#BNDPROL

***** Build field/column map

EXECUTE SUBROUTINE(BLDMAP)

***** Open service

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Load service

CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(RFIDatаS

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Create datasource

```

*****
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('CREATE DATASOURC
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** Add processes
*****
USE    BUILTIN(GET_PROCESS_LIST) WITH_ARGS(#BNDPROSTR) 1
*****
SELECTLIST NAMED(#PROLST)
*****
ADD_ENTRY TO_LIST(#PROTABLE)
*****
ENDSELECT
*****
CHANGE  FIELD(#JSMCMD) TO('PUT OBJECT(*TABLE) NAME(PROC
USE    BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** Send datasource
*****
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('SEND OBJECT(*DA
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** Close service and send the HTTP response
*****
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** SUB ROUTINES
*****
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSMMSG
*****
IF     COND('#JSMSTS *NE OK')
*****
***** Close service
*****
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****
ENDIF

```

ENDROUTINE

SUBROUTINE NAME(BLDMAP)

CHANGE FIELD(#BNDMAPS) TO(BNDPRONME)

CHANGE FIELD(#BNDMAPL) TO(PROCESS_NAME)

ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDPRODSC)

CHANGE FIELD(#BNDMAPL) TO(PROCESS_DESC)

ADD_ENTRY TO_LIST(#MAPLST)

ENDROUTINE

JSMLR02 LANSA Process Information

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#BNDMAPS) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDMAPL) TYPE(*CHAR) LENGTH(30)
DEFINE FIELD(#BNDRTNCDE) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#BNDPRONME) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFUNNME) TYPE(*CHAR) LENGTH(7)
DEFINE FIELD(#BNDFUNDSC) TYPE(*CHAR) LENGTH(40)
DEFINE FIELD(#BNDFUNFIL) TYPE(*CHAR) LENGTH(13)

DEF_LIST NAME(#MAPLST) FIELDS((#BNDMAPS) (#BNDMAPL)) TY

DEF_LIST NAME(#FUNLST) FIELDS((#BNDFUNNME) (#BNDFUNDSC

DEF_LIST NAME(#FUNTABLE) FIELDS((#BNDPRONME) (#BNDFUNN

***** Build field/column map

EXECUTE SUBROUTINE(BLDMAP)

***** Open service

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Load service - Receive PRONAME

CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(RFIDDataS

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Create data source

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('CREATE DATASOURC
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** Get functions

```
USE    BUILTIN(GET_FUNCTION_LIST) WITH_ARGS(#BNDPRONME
```

```
IF     COND('#BNDRTNCDE *EQ OK')
```

```
SELECTLIST NAMED(#FUNLST)
```

```
ADD_ENTRY TO_LIST(#FUNTABLE)
```

```
ENDSELECT
```

```
CHANGE FIELD(#JSMCMD) TO('PUT OBJECT(*TABLE) NAME(FUNC
```

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

```
ENDIF
```

***** Send data source

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('SEND OBJECT(*DA
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** Close service and send the HTTP response

```
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** SUB ROUTINES

```
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSM
```

```
IF     COND('#JSMSTS *NE OK')
```

```
***** Close service
*****
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****
ENDIF
*****
ENDROUTINE
*****
SUBROUTINE NAME(BLDMAP)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDPRONME)
CHANGE  FIELD(#BNDMAPL) TO(PROCESS_NAME)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFUNNME)
CHANGE  FIELD(#BNDMAPL) TO(FUNCTION_NAME)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFUNDSC)
CHANGE  FIELD(#BNDMAPL) TO(FUNCTION_DESC)
ADD_ENTRY TO_LIST(#MAPLST)
*****
ENDROUTINE
```

JSMLR03 LANSA Function Information

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#BNDMAPS) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDMAPL) TYPE(*CHAR) LENGTH(30)
DEFINE FIELD(#BNDRTNCDE) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#BNDPRONME) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFUNNME) TYPE(*CHAR) LENGTH(7)
DEFINE FIELD(#BNDFLDNME) TYPE(*CHAR) LENGTH(10) DECIMAL
DEFINE FIELD(#BNDFLDFIL) TYPE(*CHAR) LENGTH(122)

DEF_LIST NAME(#MAPLST) FIELDS((#BNDMAPS) (#BNDMAPL)) TYPE

DEF_LIST NAME(#FLDLST) FIELDS((#BNDFLDNME) (#BNDFLDFIL))

DEF_LIST NAME(#FLDTABLE) FIELDS((#BNDPRONME) (#BNDFUNN

***** Build field/column map

EXECUTE SUBROUTINE(BLDMAP)

***** Open service

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Load service - Receive PRONAME and FUNNAME

CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(RFIDDataS

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Create data source

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('CREATE DATASOURC
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** Get function fields

```
USE    BUILTIN(GET_FUNCTION_INFO) WITH_ARGS(#BNDPRONMI
```

```
IF     COND('#BNDRTNCDE *EQ OK')
```

```
SELECTLIST NAMED(#FLDLST)
```

```
ADD_ENTRY TO_LIST(#FLDTABLE)
```

```
ENDSELECT
```

```
CHANGE FIELD(#JSMCMD) TO('PUT OBJECT(*TABLE) NAME(FUNC
```

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

```
ENDIF
```

***** Send data source

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('SEND OBJECT(*DA
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** Close service and send the HTTP response

```
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** SUB ROUTINES

```
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSM
```

```
IF     COND('#JSMSTS *NE OK')
```

```
***** Close service
*****
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****
ENDIF
*****
ENDROUTINE
*****
SUBROUTINE NAME(BLDMAP)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDPRONME)
CHANGE  FIELD(#BNDMAPL) TO(PROCESS_NAME)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFUNNME)
CHANGE  FIELD(#BNDMAPL) TO(FUNCTION_NAME)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDNME)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_NAME)
ADD_ENTRY TO_LIST(#MAPLST)
*****
ENDROUTINE
```

JSMLR04 LANSA File List

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#BNDMAPS) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDMAPL) TYPE(*CHAR) LENGTH(30)
DEFINE FIELD(#BNDFILSTR) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFILNME) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFILLIB) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFILDSC) TYPE(*CHAR) LENGTH(40)
DEFINE FIELD(#BNDFILFIL) TYPE(*CHAR) LENGTH(10)

DEF_LIST NAME(#MAPLST) FIELDS((#BNDMAPS) (#BNDMAPL)) TY

DEF_LIST NAME(#FILLST) FIELDS((#BNDFILNME) (#BNDFILLIB) (#

DEF_LIST NAME(#FILTABLE) FIELDS((#BNDFILNME) (#BNDFILLIB)

***** Build field/column map

EXECUTE SUBROUTINE(BLDMAP)

***** Open service

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Load service

CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(RFIDDataS

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Create datasource

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('CREATE DATASOURC
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** Add files

```
USE BUILTIN(GET_PHYSICAL_LIST) WITH_ARGS(#BNDFILSTR)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

```
SELECTLIST NAMED(#FILLST)
```

```
ADD_ENTRY TO_LIST(#FILTABLE)
```

```
ENDSELECT
```

```
CHANGE FIELD(#JSMCMD) TO('PUT OBJECT(*TABLE) NAME(FILE_
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** Send datasource

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SEND OBJECT(*DA
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** Close service and send the HTTP response

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** SUB ROUTINES

```
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSMMSG
```

```
IF COND('#JSMSTS *NE OK')
```

***** Close service

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

ENDIF

ENDROUTINE

SUBROUTINE NAME(BLDMAP)

CHANGE FIELD(#BNDMAPS) TO(BNDFILNME)

CHANGE FIELD(#BNDMAPL) TO(FILE_NAME)

ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDFILLIB)

CHANGE FIELD(#BNDMAPL) TO(FILE_LIBRARY)

ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDFILDSC)

CHANGE FIELD(#BNDMAPL) TO(FILE_DESC)

ADD_ENTRY TO_LIST(#MAPLST)

ENDROUTINE

JSMLR05 LANSA File Information

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#BNDMAPS) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDMAPL) TYPE(*CHAR) LENGTH(30)
DEFINE FIELD(#BNDRTNCDE) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#BNDFILNME) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFILLIB) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFLDNME) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFLDSTS) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#BNDFLDTYP) TYPE(*CHAR) LENGTH(1)
DEFINE FIELD(#BNDFLDLEN) TYPE(*DEC) LENGTH(3) DECIMALS
DEFINE FIELD(#BNDFLDDEC) TYPE(*DEC) LENGTH(3) DECIMALS
DEFINE FIELD(#BNDFLDREF) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFLDDSC) TYPE(*CHAR) LENGTH(40)
DEFINE FIELD(#BNDLGLNME) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDLGLDSC) TYPE(*CHAR) LENGTH(40)

DEFINE FIELD(#BNDHDR1) TYPE(*CHAR) LENGTH(5)
DEFINE FIELD(#BNDHDR2) TYPE(*CHAR) LENGTH(5)
DEFINE FIELD(#BNDHDRFIL) TYPE(*CHAR) LENGTH(90)
DEFINE FIELD(#BNDLGLFIL) TYPE(*CHAR) LENGTH(40)
DEFINE FIELD(#BNDDTL1) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDDTLFIL) TYPE(*CHAR) LENGTH(40)

DEF_LIST NAME(#MAPLST) FIELDS((#BNDMAPS) (#BNDMAPL)) TY

DEF_LIST NAME(#HDRLST) FIELDS((#BNDHDR1) (#BNDHDR2) (#BN

DEF_LIST NAME(#HDRLGL) FIELDS((#BNDHDR1) (#BNDHDR2) (#BN
(#BNDLGLDSC) (#BNDLGLFIL)) TYPE(*WORKING) ENTRIES(100)

```

DEF_LIST NAME(#DTLLST) FIELDS((#BNDDTL1) (#BNDDTLFIL)) TY
*****
DEF_LIST NAME(#FLDTABLE) FIELDS((#BNDFILNME) (#BNDFILLIB
*****
DEF_LIST NAME(#KEYTABLE) FIELDS((#BNDFILNME) (#BNDFILLIE
*****
DEF_LIST NAME(#LGLTABLE) FIELDS((#BNDFILNME) (#BNDFILLIB
*****
***** Build field/column map
*****
EXECUTE SUBROUTINE(BLDMAP)
*****
***** Open service
*****
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** Load service - Receive FILNAME and FILLIB
*****
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(RFIDataS
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** Create datasource
*****
USE BUILTIN(JSM_COMMAND) WITH_ARGS('CREATE DATASOU
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** Select fields
*****
USE BUILTIN(GET_FILE_INFO) WITH_ARGS(#BNDFILNME #BND
*****
IF COND('#BNDRTNCDE *EQ OK')
*****
SELECTLIST NAMED(#DTLLST)
*****
CHANGE FIELD(#BNDFLDNME) TO(#BNDDTL1)
CHANGE FIELD(#BNDFLDLEN) TO(*ZERO)
CHANGE FIELD(#BNDFLDDEC) TO(*ZERO)

```

```

CHANGE FIELD(#BNDFLDREF) TO(*BLANK)
CHANGE FIELD(#BNDFLDTYP) TO(*BLANK)
CHANGE FIELD(#BNDFLDDSC) TO(*BLANK)
*****
USE BUILTIN(GET_FIELD) WITH_ARGS(#BNDFLDNME) TO_GET(
*****
***** OK - Defined in repository
***** ER - Defined in function
***** Sample job log message
***** Field named STATE not found in the data dictionary
*****
CHANGE FIELD(#BNDFLDSTS) TO(#BNDRTNCDE)
*****
ADD_ENTRY TO_LIST(#FLDTABLE)
*****
ENDSELECT
*****
ENDIF
*****
CHANGE FIELD(#JSMCMD) TO('PUT OBJECT(*TABLE) NAME(FILE_
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** Select keys
*****
CLR_LIST NAMED(#HDRLST)
CLR_LIST NAMED(#DTLLST)
*****
USE BUILTIN(GET_FILE_INFO) WITH_ARGS(#BNDFILNME #BND:
*****
IF COND('#BNDRTNCDE *EQ OK')
*****
SELECTLIST NAMED(#DTLLST)
*****
CHANGE FIELD(#BNDFLDNME) TO(#BNDDTL1)
ADD_ENTRY TO_LIST(#KEYTABLE)
*****
ENDSELECT
*****

```

```

CHANGE FIELD(#JSMCMD) TO('PUT OBJECT(*TABLE) NAME(FILE_
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GETI
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
ENDIF
*****
***** Select logical
*****
CLR_LIST NAMED(#HDRLST)
CLR_LIST NAMED(#DTLLST)
*****
USE BUILTIN(GET_FILE_INFO) WITH_ARGS(#BNDFILNME #BNDI
*****
IF COND('#BNDRTNCDE *EQ OK')
*****
SELECTLIST NAMED(#HDRLGL)
*****
ADD_ENTRY TO_LIST(#LGLTABLE)
*****
ENDSELECT
*****
CHANGE FIELD(#JSMCMD) TO('PUT OBJECT(*TABLE) NAME(LOGI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GETI
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
ENDIF
*****
***** Send datasource
*****
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SEND OBJECT(*DA
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** Close service and send the HTTP response
*****
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** SUB ROUTINES
*****

```

```

SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSMM
*****
IF      COND('#JSMSTS *NE OK')
*****
***** Close service
*****
USE      BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****
ENDIF
*****
ENDROUTINE
*****
SUBROUTINE NAME(BLDMAP)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFILNME)
CHANGE  FIELD(#BNDMAPL) TO(FILE_NAME)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFILLIB)
CHANGE  FIELD(#BNDMAPL) TO(FILE_LIBRARY)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDSTS)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_STATUS)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDNME)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_NAME)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDTYP)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_TYPE)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDLEN)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_LENGTH)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDDEC)

```

CHANGE FIELD(#BNDMAPL) TO(FIELD_DECIMAL)
ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDFLDREF)
CHANGE FIELD(#BNDMAPL) TO(FIELD_REF)
ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDFLDDSC)
CHANGE FIELD(#BNDMAPL) TO(FIELD_DESC)
ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDLGLNME)
CHANGE FIELD(#BNDMAPL) TO(LOGICAL_NAME)
ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDLGLLIB)
CHANGE FIELD(#BNDMAPL) TO(LOGICAL_LIBRARY)
ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDLGLDSC)
CHANGE FIELD(#BNDMAPL) TO(LOGICAL_DESC)
ADD_ENTRY TO_LIST(#MAPLST)

ENDROUTINE

JSMLR06 LANSA Field List

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#BNDMAPS) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDMAPL) TYPE(*CHAR) LENGTH(30)
DEFINE FIELD(#BNDFLDSTR) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFLDNME) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFLDDSC) TYPE(*CHAR) LENGTH(40)
DEFINE FIELD(#BNDFLDFIL) TYPE(*CHAR) LENGTH(10)

DEF_LIST NAME(#MAPLST) FIELDS((#BNDMAPS) (#BNDMAPL)) TY

DEF_LIST NAME(#FLDLST) FIELDS((#BNDFLDNME) (#BNDFLDDSC)

DEF_LIST NAME(#FLDTABLE) FIELDS((#BNDFLDNME) (#BNDFLDD

***** Build field/column map

EXECUTE SUBROUTINE(BLDMAP)

***** Open service

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Load service

CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(RFIDatS

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Create datasource

```

*****
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('CREATE DATASOURC
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** Add fields
*****
USE    BUILTIN(GET_FIELD_LIST) WITH_ARGS(#BNDFLDSTR) TO_GET
*****
SELECTLIST NAMED(#FLDLST)
*****
ADD_ENTRY TO_LIST(#FLDTABLE)
*****
ENDSELECT
*****
CHANGE  FIELD(#JSMCMD) TO('PUT OBJECT(*TABLE) NAME(FIELD)
USE    BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** Send datasource
*****
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('SEND OBJECT(*DATA)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** Close service and send the HTTP response
*****
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** SUB ROUTINES
*****
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSMMSG *
*****
IF      COND('#JSMSTS *NE OK')
*****
***** Close service
*****
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****
ENDIF

```

ENDROUTINE

SUBROUTINE NAME(BLDMAP)

CHANGE FIELD(#BNDMAPS) TO(BNDFLDSTR)

CHANGE FIELD(#BNDMAPL) TO(FIELD_START)

ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDFLDNME)

CHANGE FIELD(#BNDMAPL) TO(FIELD_NAME)

ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDFLDDSC)

CHANGE FIELD(#BNDMAPL) TO(FIELD_DESC)

ADD_ENTRY TO_LIST(#MAPLST)

ENDROUTINE

JSMLR07 LANSА Field Information

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#BNDMAPS) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDMAPL) TYPE(*CHAR) LENGTH(30)
DEFINE FIELD(#BNDRTNCDE) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#BNDFLDSTS) TYPE(*CHAR) LENGTH(2)
DEFINE FIELD(#BNDFLDNME) TYPE(*CHAR) LENGTH(10) DECIMALS
DEFINE FIELD(#BNDFLDTYP) TYPE(*CHAR) LENGTH(1)
DEFINE FIELD(#BNDFLDLEN) TYPE(*DEC) LENGTH(3) DECIMALS
DEFINE FIELD(#BNDFLDDEC) TYPE(*DEC) LENGTH(3) DECIMALS
DEFINE FIELD(#BNDFLDREF) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#BNDFLDLDDSC) TYPE(*CHAR) LENGTH(40)
DEFINE FIELD(#BNDFLDLDFIL) TYPE(*CHAR) LENGTH(122)

DEF_LIST NAME(#MAPLST) FIELDS((#BNDMAPS) (#BNDMAPL)) TY

DEF_LIST NAME(#FLDTABLE) FIELDS((#BNDFLDSTS) (#BNDFLDNM

***** Build field/column map

EXECUTE SUBROUTINE(BLDMAP)

***** Open service

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Load service - Receive FLDNAME

CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(RFIDataS
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET

```

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** Create data source
*****
USE BUILTIN(JSM_COMMAND) WITH_ARGS('CREATE DATASOURI
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
CHANGE FIELD(#BNDFLDLEN) TO(*ZERO)
CHANGE FIELD(#BNDFLDDEC) TO(*ZERO)
CHANGE FIELD(#BNDFLDREF) TO(*BLANK)
CHANGE FIELD(#BNDFLDTYP) TO(*BLANK)
CHANGE FIELD(#BNDFLDDSC) TO(*BLANK)
*****
USE BUILTIN(GET_FIELD) WITH_ARGS(#BNDFLDNME) TO_GET(
*****
***** OK - Defined in repository
***** ER - Defined in function
***** Sample job log message
***** Field named STATE not found in the data dictionary
*****
CHANGE FIELD(#BNDFLDSTS) TO(#BNDRTNCDE)
*****
***** Send data as datasource fields
*****
CHANGE FIELD(#JSMCMD) TO('PUT OBJECT(*FIELD) SERVICE_LIS
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** Send data source
*****
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SEND OBJECT(*DA
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** Close service and send the HTTP response
*****
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** SUB ROUTINES

```

```

*****
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSMM
*****
IF      COND('#JSMSTS *NE OK')
*****
***** Close service
*****
USE      BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****
ENDIF
*****
ENDROUTINE
*****
SUBROUTINE NAME(BLDMAP)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDSTS)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_STATUS)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDNME)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_NAME)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDTYP)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_TYPE)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDLEN)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_LENGTH)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDDEC)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_DECIMAL)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDREF)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_REF)
ADD_ENTRY TO_LIST(#MAPLST)
*****

```

```
CHANGE  FIELD(#BNDMAPS) TO(BNDFLDDSC)
CHANGE  FIELD(#BNDMAPL) TO(FIELD_DESC)
ADD_ENTRY TO_LIST(#MAPLST)
*****
ENDROUTINE
```

JSMLR08 LANSA SOAP Type List

```
FUNCTION OPTIONS(*DIRECT)
CHANGE FIELD(#BNDGCDE) TO("00001")
EXCHANGE FIELDS(#BNDGCDE)
CALL PROCESS(*DIRECT) FUNCTION(JSMLR0A)
```

JSMLR09 LANSO SOAP Type Information

```
FUNCTION OPTIONS(*DIRECT)
CHANGE FIELD(#BNDGCDE) TO("00001")
EXCHANGE FIELDS(#BNDGCDE)
CALL PROCESS(*DIRECT) FUNCTION(JSMLR0B)
```

JSMLR10 LANSA JSON Type List

```
FUNCTION OPTIONS(*DIRECT)
CHANGE FIELD(#BNDGCDE) TO("00002")
EXCHANGE FIELDS(#BNDGCDE)
CALL PROCESS(*DIRECT) FUNCTION(JSMLR0A)
```

JSMLR11 LANSa JSON Type Information

```
FUNCTION OPTIONS(*DIRECT)
CHANGE FIELD(#BNDGCDE) TO("00002")
EXCHANGE FIELDS(#BNDGCDE)
CALL PROCESS(*DIRECT) FUNCTION(JSMLR0B)
```

JSMLR0A LANSa Type List

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)

DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#BNDMAPS) TYPE(*CHAR) LENGTH(10)

DEFINE FIELD(#BNDMAPL) TYPE(*CHAR) LENGTH(30)

DEF_LIST NAME(#MAPLST) FIELDS((#BNDMAPS) (#BNDMAPL)) TY

DEF_LIST NAME(#TYPTABLE) FIELDS((#BNDCCDE) (#BNDCTYP) (#

***** Build field/column map

EXECUTE SUBROUTINE(BLDMAP)

***** Open service

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Load service

CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(RFIDDataS

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Create datasource

USE BUILTIN(JSM_COMMAND) WITH_ARGS('CREATE DATASOU

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** Add complex types

```

SELECT  FIELDS((#BNDCCDE) (#BNDCTYP) (#BNDCEY) (#BNDCD:
*****
ADD_ENTRY TO_LIST(#TYPTABLE)
*****
ENDSELECT
*****
CHANGE  FIELD(#JSMCMD) TO('PUT OBJECT(*TABLE) NAME(TYPE
USE     BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****
***** Send datasource
*****
USE     BUILTIN(JSM_COMMAND) WITH_ARGS('SEND OBJECT(*DA
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****
***** Close service and send the HTTP response
*****
USE     BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****
***** SUB ROUTINES
*****
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSM
*****
IF      COND('#JSMSTS *NE OK')
*****
***** Close service
*****
USE     BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****
ENDIF
*****
ENDROUTINE
*****
SUBROUTINE NAME(BLDMAP)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDCCDE)
CHANGE  FIELD(#BNDMAPL) TO(TYPE_KEY)
ADD_ENTRY TO_LIST(#MAPLST)

```

CHANGE FIELD(#BNDMAPS) TO(BNDCTYP)

CHANGE FIELD(#BNDMAPL) TO(NAME)

ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDCETY)

CHANGE FIELD(#BNDMAPL) TO(ENTITY)

ADD_ENTRY TO_LIST(#MAPLST)

CHANGE FIELD(#BNDMAPS) TO(BNDCDSC)

CHANGE FIELD(#BNDMAPL) TO(DESC)

ADD_ENTRY TO_LIST(#MAPLST)

ENDROUTINE

JSMLR0B LANSA Type Information

```
FUNCTION  OPTIONS(*DIRECT)
*****
DEFINE   FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE   FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE   FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
*****
DEFINE   FIELD(#BNDMAPS) TYPE(*CHAR) LENGTH(10)
DEFINE   FIELD(#BNDMAPL) TYPE(*CHAR) LENGTH(30)
*****
DEF_LIST NAME(#MAPLST) FIELDS((#BNDMAPS) (#BNDMAPL)) TY
*****
DEF_LIST NAME(#TYPTABLE) FIELDS((#BNDECDE) (#BNDESEQ) (#
*****
***** Build field/column map
*****
EXECUTE  SUBROUTINE(BLDMAP)
*****
***** Open service
*****
USE      BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** Load service
*****
CHANGE   FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(RFIDDataS
USE      BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** Create datasource
*****
USE      BUILTIN(JSM_COMMAND) WITH_ARGS('CREATE DATASOU
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
SELECT   FIELDS((#BNDESEQ) (#BNDEDSC) (#BNDSCDE) (#BNDCCI
*****
```

```

IF      COND('#BNDESCDE *NE *BLANK')
*****
***** Simple Type
*****
CHANGE  FIELD(#BNDCARY) TO('N')
CHANGE  FIELD(#BNDCODE) TO(*BLANK)
CHANGE  FIELD(#BNDCTYP) TO(*BLANK)
*****
FETCH   FIELDS((#BNDCDSC) (#BNDSTYP) (#BNDSARY) (#BNDSNM
*****
***** Inherit simple name
*****
IF      COND('#BNDENME *EQ *BLANK')
CHANGE  FIELD(#BNDENME) TO(#BNDSNME)
ENDIF
*****
***** Inherit simple description
*****
IF      COND('#BNDEDSC *EQ *BLANK')
CHANGE  FIELD(#BNDEDSC) TO(#BNDCDSC)
ENDIF
*****
***** Inherit simple field
*****
IF      COND('#BNDEETY *EQ *BLANK')
CHANGE  FIELD(#BNDEETY) TO(#BNDSETY)
ENDIF
*****
ENDIF
*****
IF      COND('#BNDCODE *NE *BLANK')
*****
***** Complex Type
*****
CHANGE  FIELD(#BNDSARY) TO('N')
CHANGE  FIELD(#BNDESCDE) TO(*BLANK)
CHANGE  FIELD(#BNDSTYP) TO(*BLANK)
*****
FETCH   FIELDS((#BNDCDSC) (#BNDCTYP)) FROM_FILE(BNDCTPS)

```

***** Inherit complex description

```
IF      COND('#BNDEDSC *EQ *BLANK')
CHANGE  FIELD(#BNDEDSC) TO(#BNDCDSC)
ENDIF
```

```
ENDIF
```

```
IF      COND('#BNDENME *EQ *BLANK')
CHANGE  FIELD(#BNDENME) TO('NONAME')
ENDIF
```

```
ADD_ENTRY TO_LIST(#TYPTABLE)
```

```
CHANGE  FIELD(#BNDESEQ) TO(*DEFAULT)
CHANGE  FIELD(#BNDEDSC) TO(*DEFAULT)
CHANGE  FIELD(#BNDENME) TO(*DEFAULT)
CHANGE  FIELD(#BNDEETY) TO(*DEFAULT)
```

```
CHANGE  FIELD(#BNDESCDE) TO(*DEFAULT)
CHANGE  FIELD(#BNSDSC) TO(*DEFAULT)
CHANGE  FIELD(#BNDSTYP) TO(*DEFAULT)
CHANGE  FIELD(#BNDSARY) TO(*DEFAULT)
CHANGE  FIELD(#BNDSNME) TO(*DEFAULT)
CHANGE  FIELD(#BNDSETY) TO(*DEFAULT)
```

```
CHANGE  FIELD(#BNDCODE) TO(*DEFAULT)
CHANGE  FIELD(#BNDCDSC) TO(*DEFAULT)
CHANGE  FIELD(#BNDCTYP) TO(*DEFAULT)
CHANGE  FIELD(#BNDCARY) TO(*DEFAULT)
```

```
ENDSELECT
```

```
CHANGE  FIELD(#JSMCMD) TO('PUT OBJECT(*TABLE) NAME(TYPE
USE      BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMS)
```

***** Send datasource

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('SEND OBJECT(*DA
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** Close service and send the HTTP response

```
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** SUB ROUTINES

```
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSM
```

```
IF     COND('#JSMSTS *NE OK')
```

***** Close service

```
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
ENDIF
```

```
ENDROUTINE
```

```
SUBROUTINE NAME(BLDMAP)
```

```
CHANGE FIELD(#BNDMAPS) TO(BNDECDE)
```

```
CHANGE FIELD(#BNDMAPL) TO(TYPE_KEY)
```

```
ADD_ENTRY TO_LIST(#MAPLST)
```

```
CHANGE FIELD(#BNDMAPS) TO(BNDESEQ)
```

```
CHANGE FIELD(#BNDMAPL) TO(SEQUENCE)
```

```
ADD_ENTRY TO_LIST(#MAPLST)
```

```
CHANGE FIELD(#BNDMAPS) TO(BNDENME)
```

```
CHANGE FIELD(#BNDMAPL) TO(NAME)
```

```
ADD_ENTRY TO_LIST(#MAPLST)
```

```
CHANGE FIELD(#BNDMAPS) TO(BNDEETY)
```

```
CHANGE FIELD(#BNDMAPL) TO(ENTITY)
```

```
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDEDSC)
CHANGE  FIELD(#BNDMAPL) TO(DESC)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDSCDE)
CHANGE  FIELD(#BNDMAPL) TO(SIMPLE_CODE)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDSTYP)
CHANGE  FIELD(#BNDMAPL) TO(SIMPLE_TYPE)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDSARY)
CHANGE  FIELD(#BNDMAPL) TO(SIMPLE_ARRAY)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDCCDE)
CHANGE  FIELD(#BNDMAPL) TO(COMPLEX_CODE)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDCTYP)
CHANGE  FIELD(#BNDMAPL) TO(COMPLEX_TYPE)
ADD_ENTRY TO_LIST(#MAPLST)
*****
CHANGE  FIELD(#BNDMAPS) TO(BNDCARY)
CHANGE  FIELD(#BNDMAPL) TO(COMPLEX_ARRAY)
ADD_ENTRY TO_LIST(#MAPLST)
*****
ENDROUTINE
```

7.25 Test Editor

The Test Editor is started from Studio. It is recommended that you use Studio to manage your projects and associated files.

The Test Editor is used to perform EDI processing and XML transformation tests.

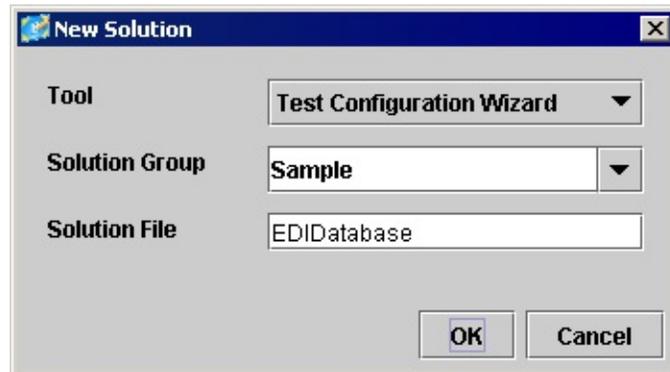
Review the following topics:

[7.25.1 Create a Test Project](#)

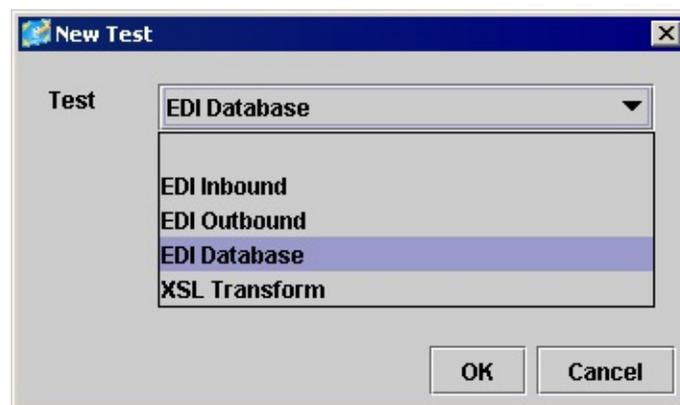
[7.25.2 Test Samples](#)

7.25.1 Create a Test Project

1. From the Project Solutions node select the *New Solution* menu item.
2. Select the *Test Configuration Wizard* tool and enter or select the *Group* folder to receive the project file.



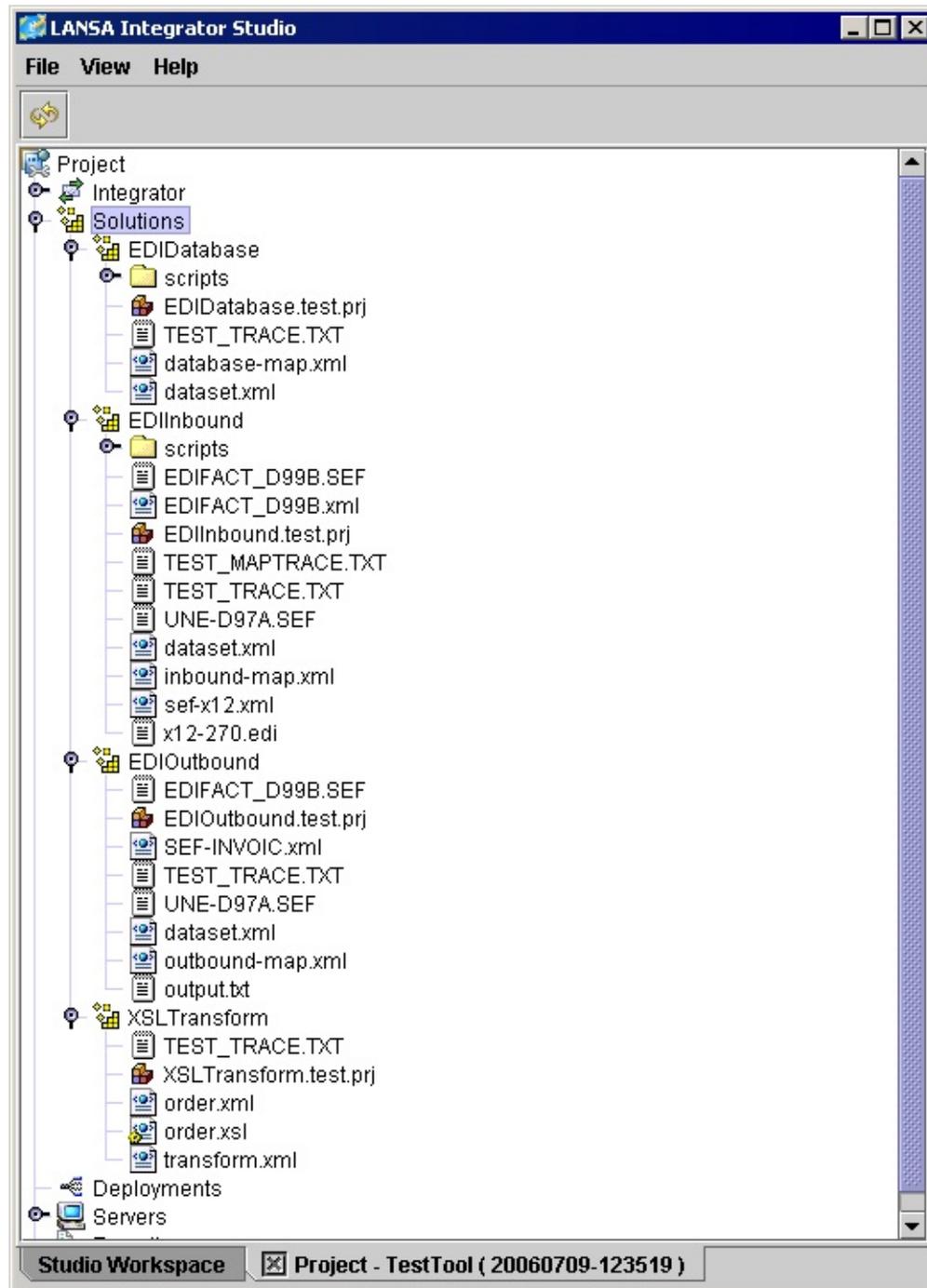
3. Enter the project File name. The extension '.test.prj' is added automatically.



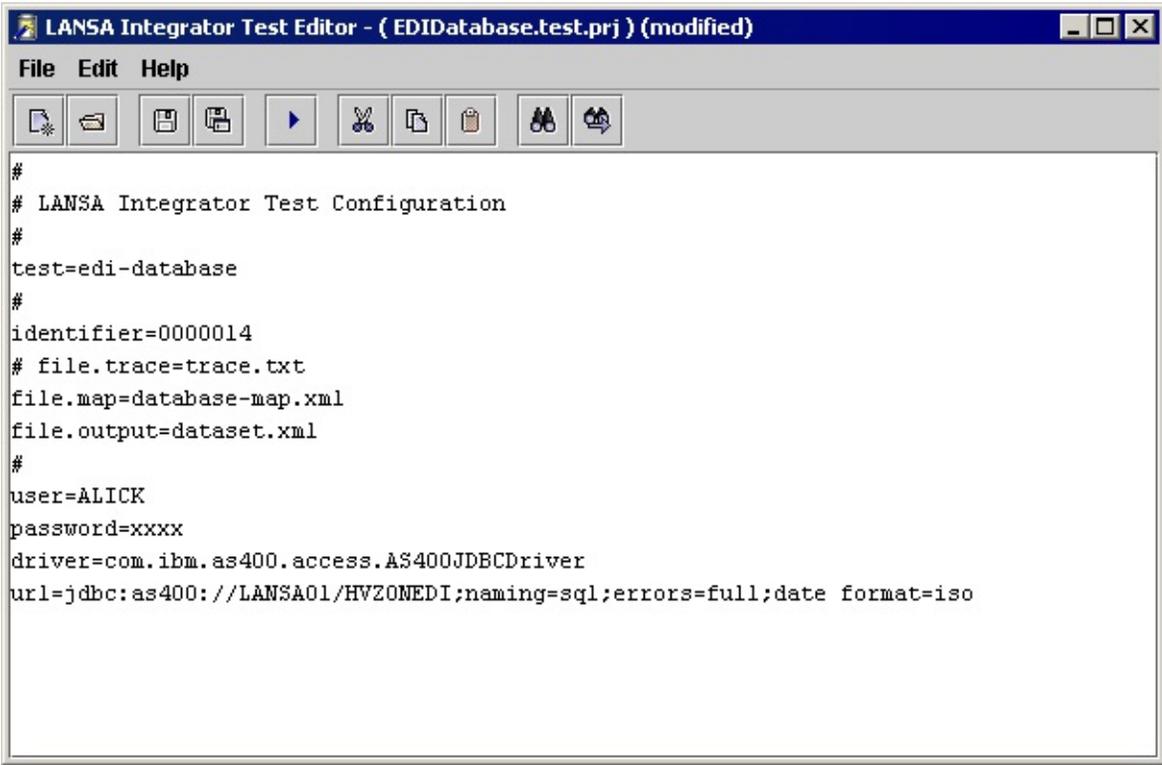
4. Select type of test.

7.25.2 Test Samples

Studio Test Samples Overview



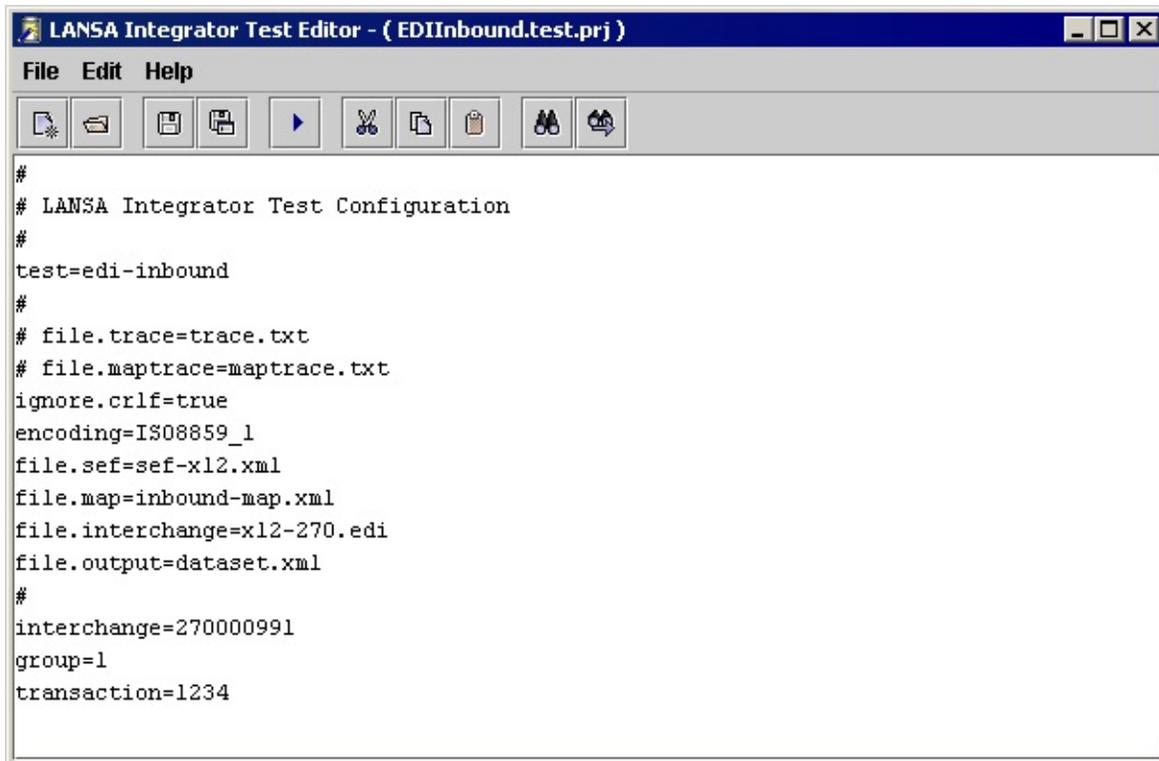
EDI Database Test



The screenshot shows a window titled "LANSA Integrator Test Editor - (EDIDatabase.test.prj) (modified)". The window has a menu bar with "File", "Edit", and "Help". Below the menu bar is a toolbar with icons for file operations (new, open, save, save as, print, copy, paste, undo, redo) and a play button. The main text area contains the following configuration text:

```
#  
# LANSA Integrator Test Configuration  
#  
test=edi-database  
#  
identifier=0000014  
# file.trace=trace.txt  
file.map=database-map.xml  
file.output=dataset.xml  
#  
user=ALICK  
password=xxxx  
driver=com.ibm.as400.access.AS400JDBCdriver  
url=jdbc:as400://LANSA01/HVZONEDI;naming=sql;errors=full;date format=iso
```

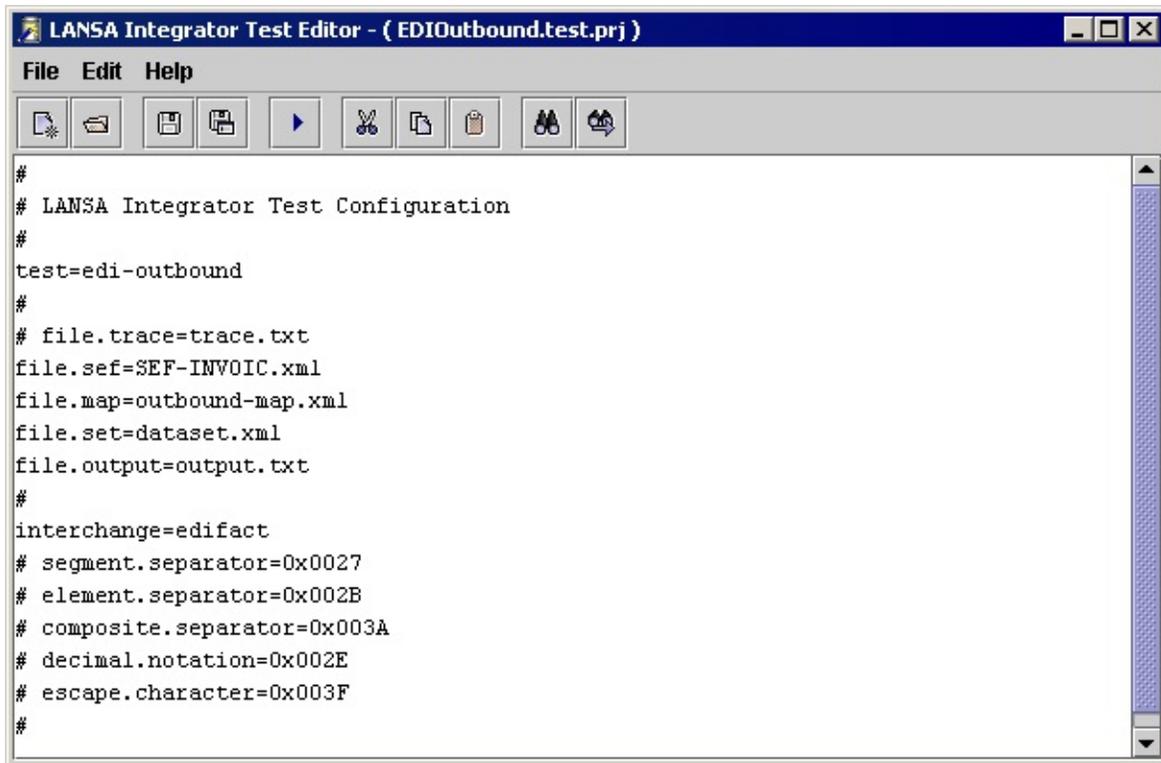
EDI Inbound Test



The screenshot shows a window titled "LANSA Integrator Test Editor - (EDIInbound.test.prj)". The window has a menu bar with "File", "Edit", and "Help". Below the menu bar is a toolbar with icons for file operations (new, open, save, save as, run, cut, copy, paste, zoom in, zoom out). The main area contains the following configuration text:

```
#
# LANSA Integrator Test Configuration
#
test=edi-inbound
#
# file.trace=trace.txt
# file.maptrace=maptrace.txt
ignore.crlf=true
encoding=ISO8859_1
file.sef=sef-x12.xml
file.map=inbound-map.xml
file.interchange=x12-270.edi
file.output=dataset.xml
#
interchange=270000991
group=1
transaction=1234
```

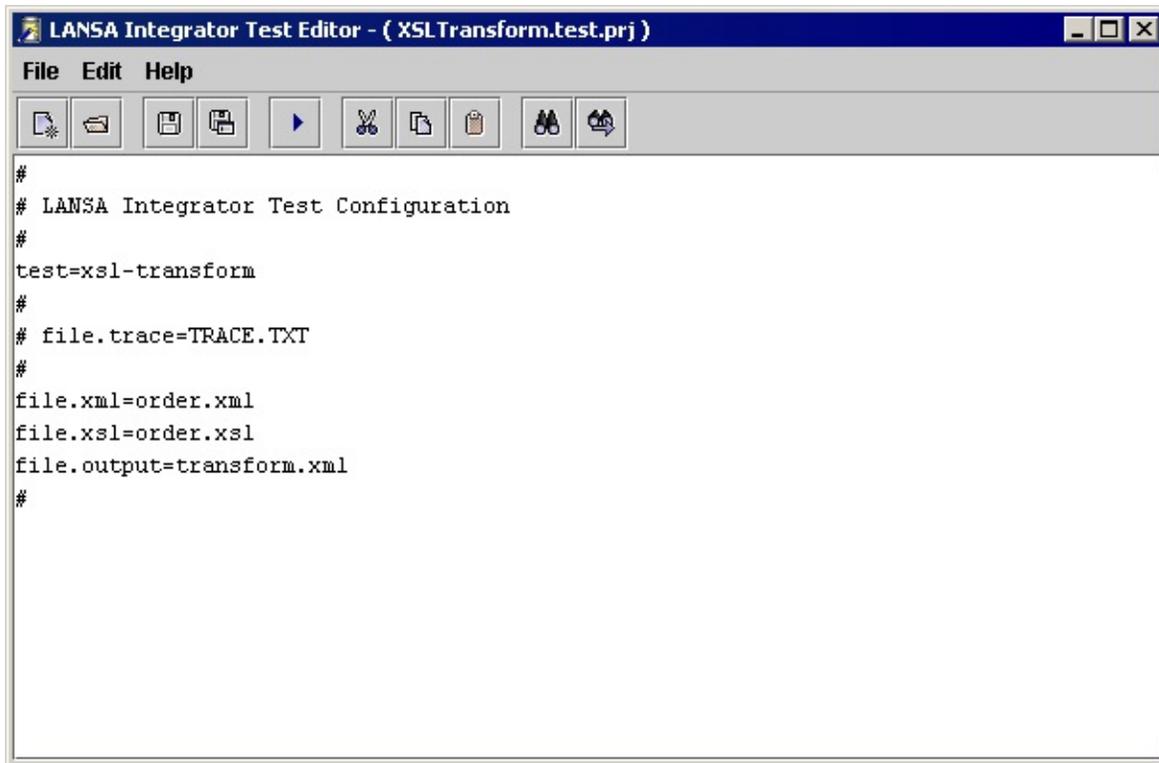
EDI Outbound Test



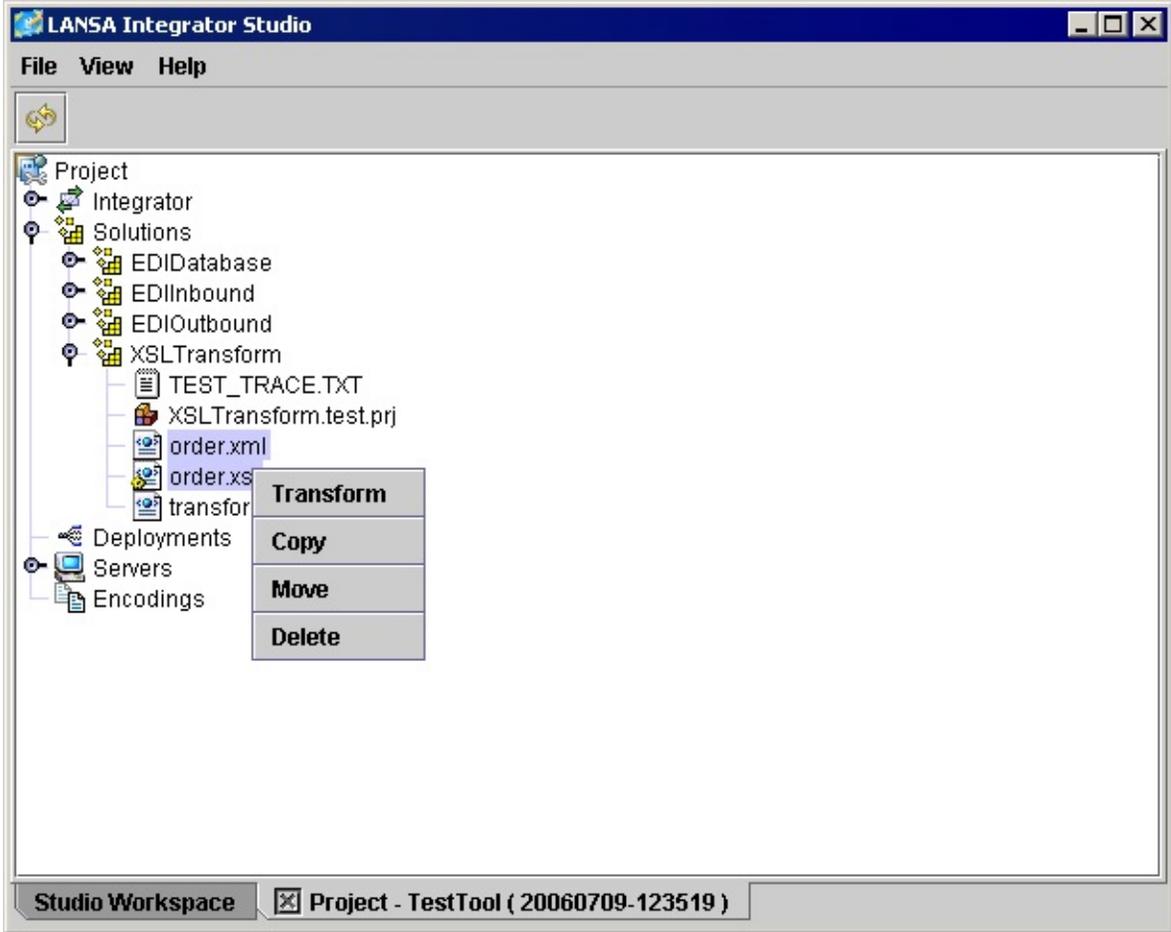
The image shows a screenshot of a software application window titled "LANSA Integrator Test Editor - (EDIOutbound.test.prj)". The window has a menu bar with "File", "Edit", and "Help". Below the menu bar is a toolbar with icons for file operations (new, open, save, save as, run, cut, copy, paste, zoom in, zoom out) and a vertical scrollbar on the right side. The main area of the window contains a text editor with the following configuration text:

```
#
# LANSA Integrator Test Configuration
#
test=edi-outbound
#
# file.trace=trace.txt
file.sef=SEF-INVOIC.xml
file.map=outbound-map.xml
file.set=dataset.xml
file.output=output.txt
#
interchange=edifact
# segment.separator=0x0027
# element.separator=0x002B
# composite.separator=0x003A
# decimal.notation=0x002E
# escape.character=0x003F
#
```

XML Transformation Test



XML transformation testing can also be done from Studio by selecting both an XML file and XSL file.



9. Remote Function Invocation

Remote Function Invocation (RFI) is a means of sending and receiving serialized java objects between a client JVM and the remote JSM server.

The Java Service Manager RFI feature is middleware that makes remote integration as easy as coding to local objects. The Java client developer creates and works with local objects and then sends these objects to the remote JSM server, the remote service uses a local copy of these objects to supply data to the host program. The host program can also create new objects and return a copy to the Java client program for local processing. Compared with JDBC, this is much faster because network traffic is minimized and it is much easier and more natural to code. RFI can be called either by LANSAs functions or 3GL programs.

Review the following topics:

[9.1 Using the RFIService client class](#)

[9.2 Working with the RFIDataSource on the client](#)

[9.3 RFI Example](#)

[9.4 RFI Example Viewer](#)

9.1 Using the RFIService client class

RFIService is the client class used for Remote Function Invocation programming.

To create an instance of RFIService use the static getInstance method. This method requires a properties parameter. These properties control the communication characteristics of the host connection.

Property	Comments
rfi.server	Host protocol and address http://your.own.url:port.
rfi.proxy	Proxy server.
rfi.user	Basic authentication user.
rfi.password	Basic authentication password.
rfi.agent	Host agent program. Default - /cgi-bin/jsmdirect.
rfi.protocol.encoding	HTTP protocol head encoding. Default - ISO8859_1.
rfi.authentication.encoding	HTTP Basic authentication encoding. Default - ISO8859_1.
rfi.server.trusted	Required for an SSL connection if the trust store does not contain the peer certificate. A value of true means that server certificate will be trusted.

Example

```
Properties properties = new Properties ();  
  
properties.put ( "rfi.server", "http://your.own.url:port" );  
  
RFIService service = RFIService.getInstance ( properties );
```

The RFIService has two methods to send and receive serialized objects. The getObject method is used to receive a serialized response object from the host server using the specified service name.

```
Object getObject ( String service )  
Object getObject ( RFIObject service ) ;
```

Example

```
Object object = service.getObject ( "EMPLOYEE-FETCH" ) ;
```

The putObject method sends a serialized object to the host server and receives a response object from this named service.

```
Object putObject ( String service, Object sendObject ) ;  
Object putObject ( RFIObject sendObject ) ;
```

Example

```
Object object = service.putObject ( "EMPLOYEE-UPDATE", employee ) ;
```

If the response status from the host is not successful then an RFIException is thrown.

If the content type of the response is not "application/x-java-serialized-object" an RFIException is thrown.

If no content was returned a null object reference is returned, else the content is de-serialized and an object reference is returned.

The RFIObject interface has one method to return the service name.

This allows the sendObject to supply the name of the service and eliminates the need for the String service name parameter.

```
public interface RFIObject  
{  
    public String getService () ;  
}
```

The RFIException class has several methods to allow access to the response status and message.

Method	Comments
int getStatus ()	Get response status code
String getMessage ()	Get response status message
byte[] getContent ()	Get response content
Enumeration propertyNames ()	Get response properties
getProperty (String property)	Get response property

9.2 Working with the RFIDataSource on the client

RFIDataSource implements the **RFIObject** interface.

Constructor	Comments
RFIDataSource (String name)	Create a new data source and service of the same name
RFIDataSource (String name, String service)	Create a new data source using specified service name

Method	Comments
String getName ()	Name of data source
String getService ()	Name of service
void setService (String service)	Set service name
void clearFields ()	Clear all fields from data source
String[] getFieldNames ()	Get data source field names
void putField (String name, String value)	Add field to data source
void removeField (String name)	Remove field from data source
boolean containsField (String name)	Data source contains field
String getFieldValue (String name)	Get value of field
void clearTables ()	Clear all tables from data source
void putTable (RFIDataTable table)	Add table to data source
void removeTable (String name)	Remove table from data source
RFIDataTable getTable (String name)	Get table from data source

boolean containsTable (String name)	Data source contains table
String[] getTableNames ()	Get data source table names
boolean equals (String name)	Data source name equivalence

RFIDDataTable

Constructor	Comments
RFIDDataTable (String name, String[] columns)	Create new table
RFIDDataTable (String name, String[] columns, int initialSize)	Create new table

Method	Comments
String getName ()	Get name of table
String[] getColumns ()	Get table column names
int getColumnCount ()	Get table column count
String getColumnName (int column)	Get table column name at specified index
int getRowCount ()	Get row count
String[] getRow (int row)	Get row of data at specified index
void clear ()	Remove all rows
void remove (int row)	Remove row at specified index
void add (String[] data)	Add a row of data
void set (int row, String[] data)	Update a row of data at specified index
void insert (int row, String[] data)	Insert a row of data at specified

	index
String getValueAt (int row, int column)	Get cell value at specified row and column
void setValueAt (int row, int column, String value)	Update cell value
boolean equals (String name)	Table name equivalence

Example of creating a client-side RFIDataSource object:

```

RFIDataSource employee = new RFIDataSource ( "EMPLOYEE" );

String[] columns = { "COMMENT", "SKILL", "GRADE", "ACQUIRED" };

RFIDataTable table = new RFIDataTable ( "SKILLS", columns );

String[] data = { "Good cobol skills", "CBL", "A", "020399" };

table.add ( data );

employee.putTable ( table );

```

9.3 RFI Example

The RFI Example illustrates how to use the RFI Service to receive a `RFIDataSource` object from the remote JSM server. The contents of this data source are displayed using the RFI Example Viewer.

```
Properties properties = new Properties ();

properties.put ( "rfi.server", "http://your.own.url:port" );

RFIService service = RFIService.getInstance ( properties );

RFIDataSource dataSource = (RFIDataSource)service.getObject ( "EMPLOYEE
FETCH" );

if ( dataSource == null )
{
    printStream.println ( "No object was returned" );

    return ;
}

if ( dataSource.equals ( "EMPLOYEE" ) )
{
    printStream.println ( "Data source fields :" );
    printStream.println ( "" );

    /*
    Access fields
    */

    String[] fields = dataSource.getFieldNames () ;

    for ( int i=0; i < fields.length; i++ )
    {
        printStream.println ( fields[i] + "\t\t" + dataSource.getFieldValue ( fields[i]
    }
}
```

```

printStream.println ( "" );

if ( dataSource.containsTable ( "SKILLS" ) )
{
    printStream.println ( "Data source table : SKILLS" );
    printStream.println ( "" );

    /*
     * Access table
     */

    RFIDDataTable table = dataSource.getTable ( "SKILLS" );

    /*
     * Access table columns
     */

    String[] columns = table.getColumns () ;

    for ( int i=0; i < columns.length; i++ )
    {
        printStream.print ( columns[i] + "\t" );
    }

    /*
     * Access table rows
     */

    int rowCount = table.getRowCount () ;

    for ( int i=0; i < rowCount; i++ )
    {
        printStream.println ( "" );

        for ( int j=0; j < columns.length; j++ )
        {
            printStream.print ( table.getValueAt ( i, j ) + "\t" );
        }
    }
}

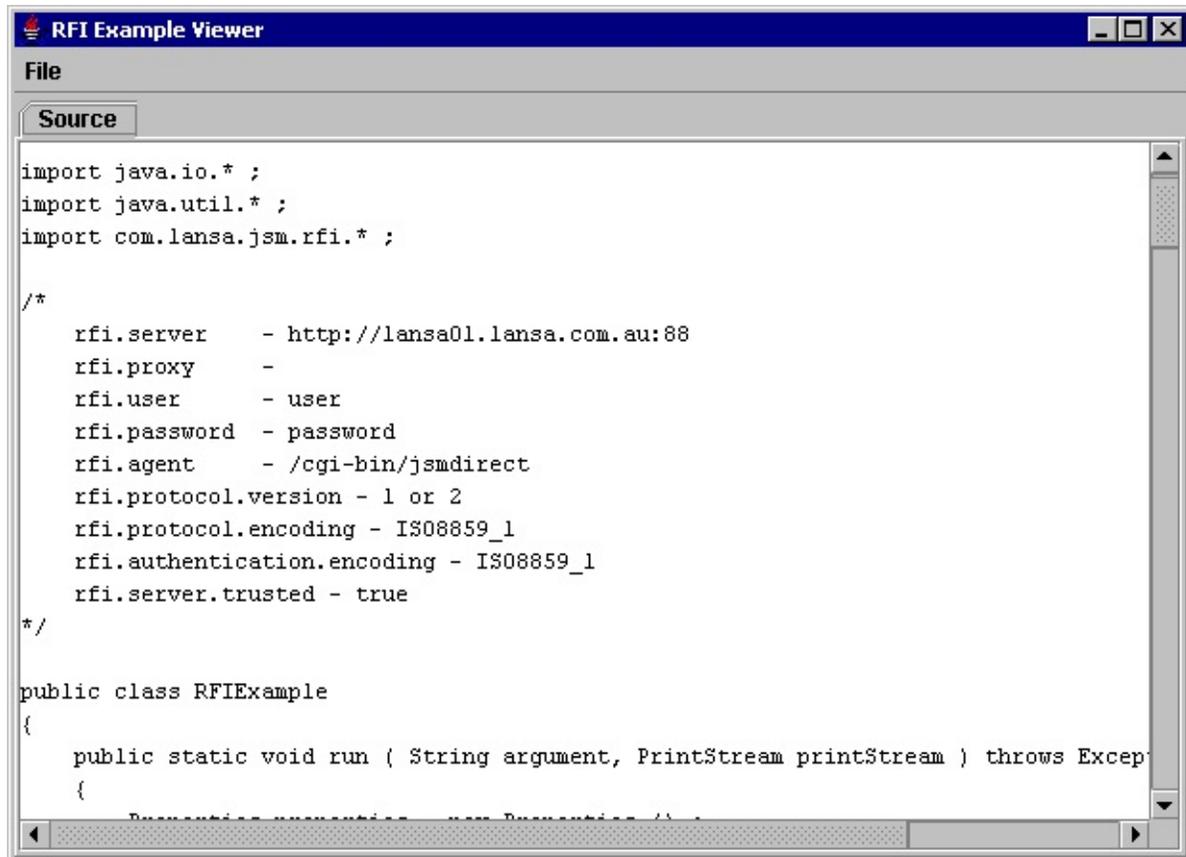
```

}
}

9.4 RFI Example Viewer

The RFI Example Viewer is a simple GUI application that illustrates how to use the RFIService client class and the JSM service RFIDataSourceService.

The Java programmer modifies the RFIExample source code and compiles and runs the RFI Example Viewer application to execute the RFIExample class to perform a Remote Function Innovation (RFI) to the remote JSM server.



The screenshot shows a window titled "RFI Example Viewer" with a menu bar containing "File". Below the menu bar is a tab labeled "Source". The main area of the window contains the following Java source code:

```
import java.io.* ;
import java.util.* ;
import com.lansa.jsm.rfi.* ;

/*
   rfi.server      - http://lansa01.lansa.com.au:88
   rfi.proxy       -
   rfi.user        - user
   rfi.password    - password
   rfi.agent       - /cgi-bin/jsmdirect
   rfi.protocol.version - 1 or 2
   rfi.protocol.encoding - ISO8859_1
   rfi.authentication.encoding - ISO8859_1
   rfi.server.trusted - true
*/

public class RFIExample
{
    public static void run ( String argument, PrintStream printStream ) throws Except
    {
        Presentation presentation = new Presentation ( ) ;
    }
}
```

10. Troubleshooting

Please review the following Troubleshooting topics before contacting your local vendor for technical support:

[10.1 How to Display the version of Java](#)

[10.2 How to Display the version of the Java Service Manager](#)

[10.3 Tracing](#)

[10.4 STRJSM command not working](#)

[10.5 Locating error messages on IBM i](#)

[10.6 JSMDirect or JSMProxy not working](#)

[10.7 Java Virtual Machine is shut down when you log off from a Windows Server](#)

10.1 How to Display the version of Java

Use the `JAVA CLASS(*VERSION)` command to display the required JDK version.

For the classic JDK set the 'java.version' property and for the IBM Technology JDK set the `JAVA_HOME` environment variable.

Default JDK

```
RMVENVVAR ENVVAR(JAVA_HOME)
JAVA CLASS(*VERSION)
```

Classic 1.4

```
RMVENVVAR ENVVAR(JAVA_HOME)
JAVA CLASS(*VERSION) PROP(('java.version' '1.4'))
```

Classic 1.5

```
RMVENVVAR ENVVAR(JAVA_HOME)
JAVA CLASS(*VERSION) PROP(('java.version' '1.5'))
```

Classic 1.6

```
RMVENVVAR ENVVAR(JAVA_HOME)
JAVA CLASS(*VERSION) PROP(('java.version' '1.6'))
```

IBM Technology 5.0 32-bit

```
ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdI
JAVA CLASS(*VERSION)
```

IBM Technology 5.0 64-bit

```
ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdI
JAVA CLASS(*VERSION)
```

IBM Technology 6.0 32-bit

```
ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdI
JAVA CLASS(*VERSION)
```

IBM Technology 6.0 64-bit

```
ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdI
JAVA CLASS(*VERSION)
```

IBM Technology 7.0 32-bit

```
ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdI
JAVA CLASS(*VERSION)
```

IBM Technology 7.0 64-bit

```
ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdI
JAVA CLASS(*VERSION)
```

IBM Technology 8.0 32-bit

```
ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdI
JAVA CLASS(*VERSION)
```

IBM Technology 8.0 64-bit

```
ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdI
JAVA CLASS(*VERSION)
```


10.2 How to Display the version of the Java Service Manager

The trace files contain the version numbers of different Integrator components. The TRANSPORT.TXT trace file contains the version number of the jsm.jar file, this value follows the manager: version entry. The version number of the DCXS882X service program follows the transport: Client-Version entry. The CCSID, language and country code of the client application is also shown.

```
===== TRACE OPEN =====
```

```
manager: version : 11.4.0
```

```
manager: date   : 2006-05-11 15:04:36 +1000
```

```
client thread: client address : 127.0.0.1
```

```
client thread: read command
```

```
transport: read protocol head
```

```
transport: Client-Type : iSeries
```

```
transport: Client-Version : 11.4.0
```

```
transport: Client-CodePage : 37
```

```
transport: Client-Language : ENU
```

```
transport: Client-Country : AU
```

```
transport: Content-Type : application/command
```

```
transport: Command-Length : 12
```

```
===== TRACE CLOSE =====
```

The HTTP_KEYWORDS.TXT file created by most HTTP services also contains version information.

The DIRECT-VERSION keyword value is the version number of JSMDirect.

The PROXY-VERSION keyword value is the version number of JSMProxy.

The USER-AGENT keyword value contains the version number of the Integrator User Agent.

```
PROXY-VERSION = 11.4.0
```

```
DIRECT-VERSION = 11.4.0
```

```
USER-AGENT = LANSA Integrator UserAgent/11.4.0 (Client)
```

```
USER-AGENT = LANSA Integrator RFIAgent/11.4.0 (Client)
```

To determine the version of JSMDirect and also JSMProxy, use a web browser to execute the CGI programs with no service argument.

For IBM i: <http://myserver/cgi-bin/jsmdirect>
 <http://myserver/cgi-bin/jsmproxy>

For Windows <http://myserver/cgi-bin/jsmdirect.exe>
 <http://myserver/cgi-bin/jsmproxy.exe>

The CGI programs will return an HTML page displaying version information.

10.3 Tracing

For details of the Tracing facilities provided with the Integrator, refer to:

[Tracing](#)

[Apache Tracing \(IBM i\)](#)

10.4 STRJSM command not working

Refer to:

[JSM Job Management](#)

10.5 Locating error messages on IBM i

Refer to:

[JSM Job Management](#)

10.6 JSMDirect or JSMProxy not working

Refer to:

[JSMDirect Program Logic](#)

[JSMProxy Program Logic](#)

10.7 Java Virtual Machine is shut down when you log off from a Windows Server

Scenario: You log on to a Server and your JSM Administrator is running. You do some work. The JSM Administrator is still running. Then you log off from the Server. The JSM Administrator stops working, which doesn't seem correct.

What happens is that the Java Virtual Machine is shut down when you log off. This is the correct behavior in most circumstances. It is not correct when it runs from a Windows Service.

How can the Java Virtual Machine tell the difference? It can't, but you can tell it not to shut down when you log off by using the `-Xrs` option.

By including the `-Xrs` option in the `jmmmgrdta.txt` file, the above scenario should no longer occur.

11. Recommendations

- **DO NOT** make a backup copy of jar files in the jar sub-directory. When JSM starts, all files in the jar directory are included in the Java classpath. It is possible for backup files to appear before the intended jar files in the Java classpath. This will cause classes in the backup files to be used instead of the classes in intended jar files.
- **DO NOT** modify entries in the LANSAs studio section of service property files, but add your own section to the bottom of these files. All property file entries must exist within a studio-project section, during an upgrade process any entries found outside a section will be placed in the studio-project id="00000000-000000" name="unassigned" section. They should be assigned to another section or deleted.
- LANSAs function fields that do not contribute to the Java service should be prefixed with JSM. This reduces the amount of data being transferred between the function and the Java service. To illustrate this recommendation, the fields #JSMSTS, #JSMMSG and #JSMCMD are used in some LANSAs Integrator examples.
- When using the MQSeries programs, fields that receive the message and correlation id must be prefixed with JSM. These fields can receive 24 bytes of data outside of the alphanumeric range. This means that they cannot be sent to the Java service because the native to Unicode conversion will fail.
- It is a matter of style, but if command keyword values are static then do not use a subroutine to build the command using a sequence of concatenations. Include the keyword directly in the command string.

Integrator Tutorials

The LANSAs Integrator Tutorials are a set of exercises designed to introduce the skills required to build Integrator applications with LANSAs using various transport protocols.

The following tutorials are included:

[INT001 - Verify JSM Install & Test](#)

[INT002 - Getting Started with Basic JSM Operations](#)

[INT003 - Using the FTP Service](#)

[INT004 - Using the LANSAs User Agent](#)

[INT005 - Department Inquiry Bindings](#)

[INT006 - Department Inquiry Functions](#)

[INT007 - Department List Inquiry](#)

[INT008 - Department & Employee Server](#)

[INT009 – Department & Employee Client \(Optional\)](#)

[INT010A - SOAP Service - Define Server](#)

[INT010B - SOAP Service - Define Agent](#)

[INT010C - SOAP Service - Test](#)

[INT011 - Create Excel Document](#)

[INT012 - Create Excel Document with Template and Formatting](#)

[INT013 - Create PDF Output using PDFDocumentService](#)

[Integrator Tutorials](#)

[Naming conventions](#)

About the Tutorials

Who Should Use the Tutorials?

A novice can use these tutorials and experienced LANSAs developers who wish to learn how to build applications with LANSAs Integrator.

In order to use the tutorials, you must be able to create LANSAs processes and functions using RDML & RDMLX. You should know how to edit RDML/X and how to create fields in the LANSAs Repository.

How Do I Use the Tutorials?

It is recommended that you complete the Tutorials in sequence. Some processes and functions created in early tutorials are reused in later tutorials.

To allow for more than one developer to use the tutorials, all LANSAs object names will be prefixed with iii. You may use any three characters, such as the initials of your name, for the iii characters. For example, if your name is John David Smith you can use the characters **JDS**. When asked to create a function named **iiiFN01**, you will create a function named **JDSFN01**. Always remember to replace **iii** with your unique 3 characters.

To check that your LANSAs partition is properly set up to use the tutorials, refer to the Tutorial Installation (following).

What Partition Should I Use?

It is recommended that you use the DEM partition for the tutorial. The DEM is automatically installed with LANSAs for iSeries and Visual LANSAs systems. The DEM partition contains the Personnel System demonstration and all required files used by the tutorial.

If you do not use the DEM partition, you can set up another partition with the Personnel System files.

The partition you use must be RDMLX-enabled.

Tutorial Installation

In order to use the LANSAs Integrator Tutorials, you must install the SET materials. Refer to the SET Guide for details about installing SET.

You must also install Studio and the User Agent. For further information, refer to [Task: Install Other Features](#) or [Upgrade LANSAs on Windows](#) in the *Installing LANSAs on Windows Guide*.

Some tutorials are run using JSMDIRECT which requires an installed HTTP

Server. Refer to [JSMDIRECT](#) for information.

INT002 (and following tutorials) use a LANSAs templates JSMXSKEL and BBJSMXCMD. These are not shipped with LANSAs and you must create them. Instructions for doing so and the necessary code are supplied in [Create RDMLX Templates](#).

How Many Developers Can Use the Training?

There is no limit on the number of developers who may use the training at the same time. However, it is important that each developer has a unique identifier for their work.

In the tutorial, each developer will use an object prefix **iii** that can be based on your initials or could be assigned by a system coordinator.

Structure of the exercises

- The first steps in an exercise will provide very precise descriptions of the tasks to be performed. As the steps and course progresses, the instructions will become much more general.
- Check off each step in the exercise as you complete it.
- Be very careful when you enter the names of the different files and processes.
- Complete only the requested tasks.

Your Feedback

- Your feedback regarding these tutorials will help us improve the overall quality of the LANSAs documentation and training. Please email your comments to lansatraining@LANSAs.com.au

Naming conventions

The following naming standards are used throughout this training course for RDMLX processes and functions, projects created using the Wizard, XML files and jar files. The naming standard will help you to relate a project with the XML it uses and the binding jar files generated from it, as well as to which functions the bindings are referred.

Process names: **iiiPROnn**

where

iii is your unique 3 characters

nn is a sequential number

Function names: **iiiFNnn**

where

iii is your unique 3 characters

nn is a sequential number

Projects: **processname_action.xml.prj**

where

processname is the name of the process that groups the client and server RDMLX functions

action is either **request** or **response**

XML files: **processname_action.xml**

where

processname is the name of the process that groups the client and server RDMLX functions

action is either **request** or **response**

INT001 - Verify JSM Install & Test

You can only complete this first tutorial if you have Visual LANSA installed.

Objectives:

- To become familiar with the SET materials.
- To learn different ways of finding a specific SET Example.
- To execute the SET218G example, which transfers a list of files from FTP.
- To locate the RDMLX source code of the FTP function.
- To highlight the advantages of using the FTP Service.

To achieve these objectives, you will complete the following:

- [Step 1. Find a specific SET example](#)
- [Step 2. Execute SET218 FTP example](#)
- [Step 3. Locating Source Code](#)
- [Summary](#)

Before You Begin

This exercise uses a SET example which should be installed and compiled on the server running the Java Services Manager (usually your IBM i server). The simplest way to install the required material is to import the SET examples from the files available on the LANSA web site.

See: <http://www.lansa.com/support/set/index.htm>

This is the recommended approach.

Alternatively, refer to the SET examples in the online guide as detailed below and create all the repository objects required and then compile the processes and functions. Note that to execute the SET_218 example the following processes and functions, plus all associated files, fields and system variables are required:

Process: SET_MOD, S_SYSVAR and SET_218.

You will find further details in the SET guide, as explained in [Step 1. Find a specific SET example](#) .

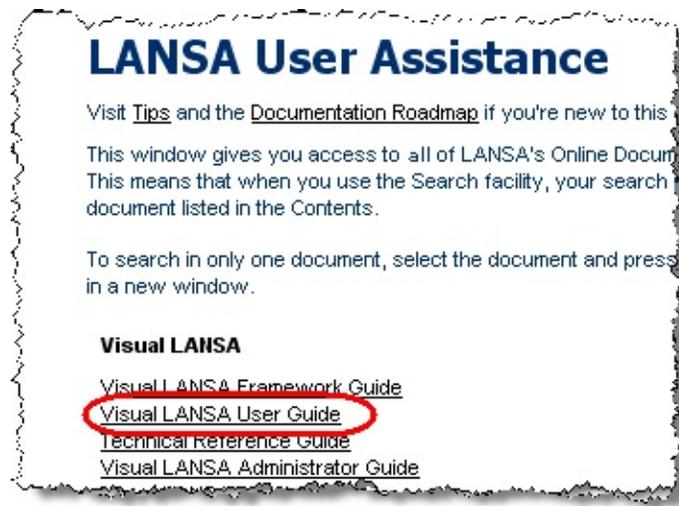
Step 1. Find a specific SET example

In this step, you will open the online guide, locate the SET manual and find a specific LANSAs Integrator example. In this case, you are searching for a LANSAs Integrator example that explains how to use the file transfer protocol using the FTP Service.

1. If you don't already have them, download the required SET files from the Support pages of the [LANSAs web site](#). Now go to point 2 Expand the items in the *Contents* tab of *The SET Collection*.

Alternatively:

1. If you have the LANSAs online documentation, you will find the SET files there.
 - a. Open the LANSAs Documentation from the Windows *Start* menu, by clicking on the *Documentation* item in your LANSAs program list.
 - b. In the *Documentation Directory*, select the **Global Guide (All guides)** and from the front page of the *LANSAs User Assistance* that opens, click on the *Visual LANSAs User Guide*.



- c. When the first page of the *Visual LANSAs User Guide* opens, select the  SET icon in the toolbar to open the *SET (Samples, Examples & Templates) Collection*.
2. Expand the items in the *Contents* tab of *The SET Collection*.
 3. Select the *Index* tab.
 4. Enter the keyword 'FTP' to find the FTP example.

Press the *Display* button or double click the FTP entry, and then select the topic you require, in this case *Transfer File(s) using FTP*.

5. Select the *Contents* tab. Notice that the selected topic is highlighted in the *Contents* list. This selection will only occur if the *Contents* list is open (i.e. expanded).

6. Select the *Search* tab.

Try to find the SET example using different search words. Use quotes if you wish to search for more than one word (e.g. a phrase) such as "**using FTP**". You must enter whole words or else use the * wild card in place of the missing letters. If you are looking for a heading, select (ü) the *Search titles only* option.

9. Double click on the entry you are looking for to display it in the window.

10. Select the *Contents* tab again. Notice that the selected topic is highlighted in the *Contents* list.

Step 2. Execute SET218 FTP example

In this step, you will read through the documentation for the SET218 example and execute it. The SET materials must have already been imported into your partition and the Java Service Manager already started.

1. Locate the main page of SET218 Using the JSM and FTP example.
2. Review the screen image and read the text including the notes and suggestions.
3. Click on the *To Execute this Example* hyperlink at the top of the page.
4. Get a list of files from an FTP folder by executing example 218G.

```
SET218G69          5250 User Interface to FTP

Type in FTP Server, User and Password and fully qualified path
transfer. The FTP Action can be PUT or GET.

*FROMFILE means put the files in the same place as the From File

Ftp User          T11PGMLIB
Ftp Password

Ftp Host          EARTH

FTP
Action           From File Path
PUT            /TMP/DEBUG.LOG
To /TEMP/DEBUG.LOG
PUT
To *FROMFILE
PUT
To *FROMFILE

F1=Help  F3=Exit  F12=Cancel  F14=Messages
```

The above example connects to an IBM i server "EARTH". The FTP, PUTs the file DEBUG.LOG from the /TMP folder in the root of the IFS to the folder /TEMP, also in the root of the IFS.

Note: Your network administrator should provide you with the information required for the IP Address, FTP Server, and a User Id and Password if

required. Refer to *Tips & Techniques* in the [Summary](#).

Step 3. Locating Source Code

In this step, you will locate the RDMLX source code of the function you executed in the previous step.

1. Locate the main page of SET218 Using the JSM and FTP example.
2. Click on The Things that Make Up this Example hyperlink at the top of the page.
3. Click on the hyperlink for the source code of the function SET218G.

Summary

Important Observations

- There are several different ways to find an example in the SET online guide.
- The online guide is useful to decide whether the functionality of a specific example can be applied to your particular needs. The SET material is shipped in executable form with all the necessary RDMLX code. It offers readymade functionality complete with the RDMLX. It is recommended that you import the SET material and examine the code with the appropriate editor.

Tips & Techniques

- You can test the SET218G example using many public FTP sites such as ftp.mcafee.com or ftp.norton.com. Many of these public sites have a userid/password of anonymous/guest. To identify the IP Address of a FTP site, try using the PING command. For example, ping ftp.norton.com. Remember to test if the public site is operational by using your browser.
- It is good practice to separate functionality from user interface. In example 218A, SET218W function provides the Web user interface, but it is the 218A function that uses JSM to transfer a list of files using FTP. This means that if you provide SET218A with the necessary fields and lists, you could create your own User Interface and use SET218A unchanged to transfer files. The same principle applies to the other JSM examples.

What I Should Know

- Which materials are included with a SET example.
- Why the SET materials are important to you.
- How to use the SET online guide.
- How to use FTP over the Internet using the LANSA Integrator's FTP services.

INT002 - Getting Started with Basic JSM Operations

Objectives:

- To create a test function, which opens a JSM session, loads a service, unloads the service and then closes the JSM session. (This test function will not execute any services's commands.) A template will be used to create the test function.
- To learn the basic structure of the JSM Built-in Functions and how they are used together.
- To learn how to Open and Close JSM.
- To learn how to Load and Unload a service.
- To learn how to handle errors in the JSM.

To achieve these objectives, you will complete the following:

- [Step 1. Create a Function using Template JSMXSKELE](#)
- [Step 2. Add a Working list to save Status and Messages to the CHECK_STS Subroutine](#)
- [Step 3. Create RDML function to display statuses and messages](#)
- [Step 4. Test iiiFN01 function](#)
- [Summary](#)

Step 1. Create a Function using Template JSMXSKEL

In this step, you will create a test function using the template JSMXSKEL to Open and Close the JSM and load the FTPService service. If you do not have a copy of JSMXSKEL, you must first create it using the instructions in [Create RDMLX Templates](#).

1. Using the LANSAs development environment, sign on to the partition nominated for the tutorials (usually DEM).
2. Check if fields JSMXCMD, JSMXHDLE1, JSMSTS and JSMMSG are defined in the Repository as these fields are required by the JSMXSKEL template. If these fields do not exist, you may create them as follows:
 - JSMXCMD STRING(65535,0) (Must be an RDMLX field)
 - JSMXHDLE1 ALPHA(4,0)
 - JSMXSTS ALPHA(20,0)
 - JSMXMSG ALPHA(200,0)
3. Create a new LANSAs process named iiiPRO01 JSM Test Process, where iii is your unique 3 characters. (If the process already exists, select a different set of characters for iii.)
4. Create a new function named iiiFN01 JSM Open/Close, belonging to process iiiPRO01. Make sure that the 'Enabled for RDMLX' checkbox is checked. Specify that the function is to be generated from an application template (from the 'Template' dropdown, select JSMXSKEL).
5. Answer the template question as shown in the table below.

Question	Answer	Comments
Do you wish to load a JSM Service?	FTPSERVICE	

6. Edit your iiiFN01 function and examine the generated RDMLX code.

The function might appear something like the following:

```
* =====  
* Process .....: JMIPRO01  
* Function .....: JMIFN01
```

* Created on: 08/11/13 at 14:27:55

* Description: JSM Open/Close

* Template.....: JSMXSKEL

* =====

FUNCTION OPTIONS(*DIRECT)

* OPEN JSM AND VERIFY STATUS

USE BUILTIN(JSMX_OPEN) TO_GET(#JSMSTS #JSMMSG
#JSMXHDLE1)

EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXHDLE1)

* BUILD THE SERVICE LOAD COMMAND

#JSMXCMD := 'SERVICE_LOAD'

EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD
'SERVICE' 'FTPSERVICE')

USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXHDLE1)

* YOUR OWN LOGIC HERE

*

* UNLOAD SERVICE

#JSMXCMD := 'SERVICE_UNLOAD'

USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXHDLE1)

* CLOSE JSM AND VERIFY STATUS

USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1)
TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXHDLE1)

*

RETURN

* Subroutine to build JSM commands. existing JSM command
SUBROUTINE NAME(KEYWRD) PARMs((#W_CMDX *BOTH)
(#W_KEYWRD *RECEIVED) (#W_KEYVAL *RECEIVED))

DEFINE FIELD(#W_CMDX) REFFLD(#JSMXCMD)

DEFINE FIELD(#W_KEYWRD) REFFLD(#STD_TEXT)

DEFINE FIELD(#W_KEYVAL) REFFLD(#STD_TEXTL)

#W_CMDX += ' ' + #W_KEYWRD + '(' + #W_KEYVAL + ')'

ENDROUTINE

* Check the status of the JSM command issued

*

```

SUBROUTINE NAME(CHECK_STS) PARS(#W_HDLE)
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132)
DEFINE FIELD(#W_HDLE) TYPE(*CHAR) LENGTH(4)
IF COND('#JSMSTS *NE OK')
#MSGDTA := 'Error Status Code: ' + #JSMSTS
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
#MSGDTA := 'Error Message: ' + #JSMMSG
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
ENDIF
ENDROUTINE

```

Notice the commands used to Open JSMX, Load the FTPService Service, Unload the Service and Close JSMX.

- Notice the use of the KEYWRD subroutine for an easy way to build the JSM command keywords and their values. Details of the KEYWRD subroutine are explained in [INT003 - Using the FTP Service](#) .
- Notice the use of the CHECK_STS subroutine after each JSM command for error handling. This routine simply formats the error message. It does not ABORT the function when an error occurs.
- The CHECK_STS subroutine requires a parameter that specifies Handle to check status on, since multiple connections to the JSM server could be open at the same time.

Step 2. Add a Working list to save Status and Messages to the CHECK_STS Subroutine

In this step, you will define a working list to store the status and message after executing each JSM statement to the CHECK_STS subroutine to indicate the function status as the JSM performs different operations.

1. Since this is an RDMLX function, it cannot use a DISPLAY command or browse list. In the next step you will create an RDML function to call iiiFN01 and receive a working list containing messages. Define a working list WL_MSGS containing fields JSMSTS and JSMMSG, at the end of the CHECK_STS subroutine and add entries to it.

The RDMLX code might appear as follows:

```
SUBROUTINE NAME(CHECK_STS) PARMS(#W_HDLE)

*
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132)
DEFINE FIELD(#W_HDLE) TYPE(*CHAR) LENGTH(4)
*
IF COND('#JSMSTS *NE OK')
*
#MSGDTA := 'Error Status Code: ' + #JSMSTS
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
#MSGDTA := 'Error Message: ' + #JSMMSG
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
ENDIF
*
DEF_LIST NAME(#WL_MSGS) FIELDS(#JSMSTS #JSMMSG) TYPE(*WOR
ADD_ENTRY TO_LIST(#WL_MSGS)
ENDROUTINE
```

2. In order to be able to call function iiiFN01 and pass and receive a working list, change the 'Function' command as follows

```
FUNCTION OPTIONS(*DIRECT) RCV_LIST(#WL_MSGS)
```

3. Save and compile function iiiFN01.

Step 3. Create RDML function to display statuses and messages

In this step, you will create the RDML function needed to display the messages stored in the working list when executing JSM commands.

1. Create a new function named iiiFN02 JSM Display, belonging to process iiiPRO01, where iii is your unique 3 characters. Make sure that the 'Enabled for RDMLX' checkbox is NOT checked. You do not need to specify a template.
2. Create a function which will call iiiFN01, passing the working list WL_MSGS and then display the results in a browse list. Your code should look something like the following:

```
* =====  
* Copyright .....: (C) LANSAS, 1989  
* Process .....: IIIPRO01  
* Function .....: IIIFN02  
* Created by .....: PCXUSER  
* Created on .....: 08/06/20 at 14:38:31  
* Description ...: JSM Display  
* =====
```

```
* List definition to show messages
```

```
DEF_LIST NAME(#WL_MSGS) FIELDS(#JSMSTS #JSMMSG) TYPE(*W  
DEF_LIST NAME(#BL_MSGS) FIELDS(#JSMSTS #JSMMSG)
```

```
FUNCTION OPTIONS(*DIRECT)
```

```
CALL PROCESS(*DIRECT) FUNCTION(iiiFN01) PASS_LST(#WL_MSGS
```

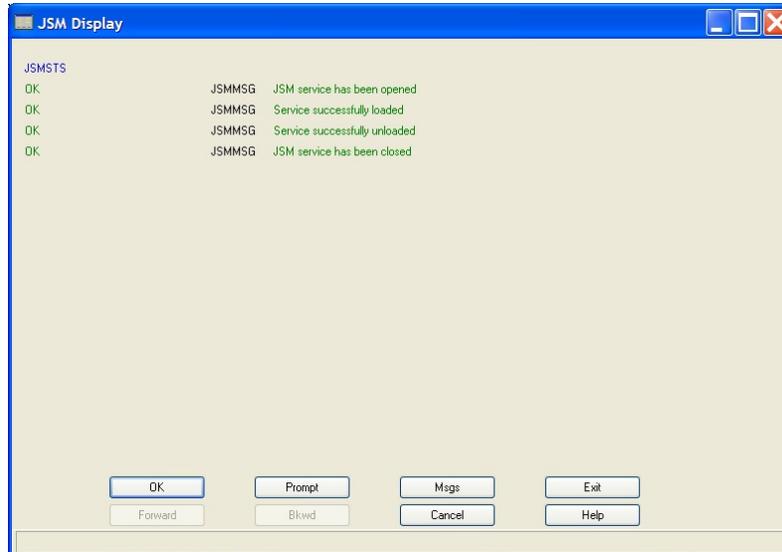
```
SELECTLIST NAMED(#WL_MSGS)  
ADD_ENTRY TO_LIST(#BL_MSGS)  
ENDSELECT
```

```
DISPLAY BROWSELIST(#BL_MSGS)
```

3. Save and compile your iiiFN02 function.

Step 4. Test iiiFN01 function

1. Verify your iiiFN01 and iiiFN02 functions have compiled successfully.
2. Execute your iiiFN02 function.
3. Review the status and messages of the program after each execution of a JSM command.



4. Close the form when finished.

Summary

Important Observations

- You can Load as many services as you wish in one function. You can also load more than one service in memory at the same time, as long as you use a different handle field.
- The action to take when a JSM error occurs depends heavily on the functionality of the application. Essentially, you should apply the same error handling methodology as the one you normally apply when writing any type of application. Usually you might decide to abort processing when the JSM error is unrecoverable. When user input is likely to have caused the error you should probably route the error back to the User Interface program.
- RDMLX functions cannot use the DISPLAY command and therefore cannot be used to display in a windows form or IBM i green screen. In these cases, a wrapper RDMLX function needs to be created in order to handle the display of information.

Tips & Techniques

- Use the template JSMXSKELE to create a function with the basic RDMLX code required by a function using the JSM. Once you become familiar with JSMXSKELE, you might copy and modify the template to suit your own specific needs.
- In the JSMXSKELE template code, the CHECK_STS subroutine simply formats the error message. It does not ABORT the function when an error occurs. You must write your own code for processing the error. You may wish to end the function, or you may want to return to an input screen to allow the user to correct an input value.
- Always write the code for closing JSM. If you use the JSMXSKELE template, this code is added for you.
- Except for HTTP server functions, always write the code for unloading JSM. If you use the JSMXSKELE template, this code is added for you.

What I Should Know

- How to use the template JSMXSKELE to create the basic RDMLX code skeleton required by a function using JSM.
- How to use the JSM Built-In Functions to control the JSM.
- How to Open and Close JSM.

- How to Load and Unload a Service.
- How to handle JSM errors by modifying the check status subroutine.

INT003 - Using the FTP Service

Objectives:

- To create a function similar to the SET218G FTP Function in your SET Examples.
- To connect to a FTP site and get the list of files from a FTP site by using the LANSA Integrator FTP service.
- To learn how to use JSM commands to interact with a loaded service.
- To use the JSMXSKEL template to write a function shell for your JSM functions.
- To use the BBJSMXCMD template to write the RDMLX code to use the JSM commands.

To achieve these objectives, you will complete the following:

- [Step 1. Build the Basic JSM functions](#)
- [Step 2. Build the JSM Commands for the FTP Service](#)
- [Step 3. Test iiiFN03 and iiiFN04 functions](#)
- [Summary](#)

Step 1. Build the Basic JSM functions

In this step, you will create a function which contains the basic RDMLX code required for your JSM functions. This function will be created using the JSMXSKEL template. You will build your function using repository fields which were imported for the SET218 example.

1. Create a new LANSAPROCESS named iiiPRO02 JSM Process, where iii is your unique 3 characters. (If the process already exists, select a different set of characters for iii.).
2. Create a new function named iiiFN03, belonging to process iiiFN02. The function will retrieve a list of files using the FTP Service. Make sure the *Enabled for RDMLX* option is selected. Choose **JSMXSKEL** as your template.
3. Answer the template questions as shown in the table following:

Question	Answer	Comments
Do you wish to load a JSM Service?	FTPSERVICE	

4. At the beginning of your function, define two fields - W_TYPE defined as an Alpha field, length 1 and W_FILE defined as an Alpha field, length 80. These fields are required by the List JSM command of the FTP service.

Your RDMLX code might appear as follows:

* Working Fields

```
DEFINE FIELD(#W_TYPE) TYPE(*CHAR) LENGTH(1) DESC('Type of file')
DEFINE FIELD(#W_FILE) TYPE(*CHAR) LENGTH(80) COLHDG('File Name')
```

5. Right after the working field definitions, define a working list which will be used to receive the list of files returned by the FTP service.
 - The working list should be named WL_218.
 - Make sure to increase the default value for the ENTRIES parameter to, say, **500**.
 - Use fields W_TYPE and W_FILE as the working list entry fields.

Your RDMLX code might appear as follows:

* Working list to receive list of files

```
DEF_LIST NAME(#WL_218) FIELDS(#W_TYPE #W_FILE) TYPE(*WORD)
```

6. Specify WL_218 as a receive list in the Function statement:

```
FUNCTION OPTIONS(*DIRECT) RCV_LIST(#WL_218)
```

7. Save function iiiFN03

8. Create a new function named **iiiFN04 Display FTP Service, belonging to process iiiPRO02**, where iii is your unique 3 characters. Make sure the 'Enabled for RDMLX' box is **NOT** checked.,Do not choose a template.

9. Copy the definitions of fields W_TYPE and W_FILE and working list WL_218 from iiiFN03.

10. Define a browse list which will be used to display the list of files returned in W_218.

- The browse list should be named BL_218.
- Use the field W_FILE as the only browse list entry field.

Your code might appear as follows:

```
DEFINE FIELD(#W_TYPE) TYPE(*CHAR) LENGTH(1) DESC('Type of file')
DEFINE FIELD(#W_FILE) TYPE(*CHAR) LENGTH(80) COLHDG('File Name')
```

```
DEF_LIST NAME(#WL_218) FIELDS(#W_TYPE #W_FILE) TYPE(*WORD)
DEF_LIST NAME(#BL_218) FIELDS(#W_FILE) ENTRIES(500)
```

11. Create a GROUP_BY for the input fields (S_218HOST, S_218DIR, S_218USER, S_218PSWD) to appear on the screen panel. (These fields were imported with the SET Examples)

Your RDMLX code might appear as follows:

* Group for all input fields

```
GROUP_BY NAME(#GB_PNLDTA) FIELDS(#S_218HOST #S_218DIR #S_218USER #S_218PSWD)
```

If field S_218DIR is not defined, define it in your function as *CHAR, length 200:

```
DEFINE FIELD(#S_218DIR) TYPE(*CHAR) LENGTH(200) DESC('FTP Directory')
```

If you needed to define S_218DIR in iiiFN04, copy this definition back to

iiiFN03 as well.

12. After the function definitions, add the following logic to the function:

- a. Insert BEGIN_LOOP and END_LOOP commands.
- b. Inside the loop, insert a REQUEST command to request the GB_PNLDTA group of fields and include the browse list BL_218.
- c. Clear both the working and browse list.
- d. Exchange fields Group GB_PNLDTA.
- e. Call iiiFN03, passing the working list WL_218.
- f. Loop through the working list and add entries to the browselist

Your code might appear as follows:

```
BEGIN_LOOP
REQUEST FIELDS(#GB_PNLDTA) BROWSELIST(#BL_218)
CLR_LIST NAMED(#WL_218)
CLR_LIST NAMED(#BL_218)
EXCHANGE FIELDS(#GB_PNLDTA)
CALL PROCESS(*DIRECT) FUNCTION(IIIFN03) PASS_LST(#WL_218)
SELECTLIST NAMED(#WL_218)
ADD_ENTRY TO_LIST(#BL_218)
ENDSELECT
END_LOOP
```

13. Compile function iiiFN04.

Step 2. Build the JSM Commands for the FTP Service

In this step, working with function iiiFN03, you will use the template BBJSMXCMD to build the JSM commands to perform the following tasks using the FTP service:

- Connect to a FTP site
- Login with a userid and password
- Set the directory
- List the files in the path specified
- Quit the FTP site.

Using the BBJSMXCMD template, you will use the following command, keywords and values:

Command	Keyword	Value	Working List
CONNECT	HOST	#S_218HOST	
LOGIN	USER	#S_218USER	
	PASSWORD	#S_218PSWD	
CHGDIR	PATH	#S_218DIR	
LIST	DIR		WL_218
QUIT			

1. Working with your iiiFN03 function, execute a template BBJSMXCMD:
 - a. Before you begin position the cursor at the start of the comment line after the YOUR OWN LOGIC HERE comment, then on the *Design* ribbon, click on the  *Template Wizard* button.
 - b. From the list of templates select **BBJSMXCMD**.
 - c. Answer the template questions as shown in the table following:

Question	Answer	Comments
Use this template to build a	Press Enter to continue	This is an

command in a function generated by JSMXSKEL.		informative panel.
Type in the JSM command you want to build.	CONNECT	
How many keywords do you want to specify?	1	
Type in the keyword name.	HOST	
Type in the value for the HOST keyword.	#S_218HOST	The Host value entered on the screen.
Type in the name of a working list if this command uses one.	Press ENTER	No working list is required by this command
Do you wish to enter another JSM command?	Y	
Type in the JSM command you want to build.	LOGIN	
How many keywords do you want to specify?	2	
Type in the keyword name.	USER	
Type in the value for the USER keyword.	#S_218USER	The User value entered on the screen.
Type in the keyword name.	PASSWORD	
Type in the value for the PASSWORD keyword.	#S_218PSWD	The Password value entered on the screen.
Type in the name of a working list if this command uses one.	Press ENTER	No working list is required by this command
Do you wish to enter	Y	

another JSM command?		
Type in the JSM command you want to build.	CHGDIR	
How many keywords do you want to specify?	1	
Type in the keyword name.	PATH	
Type in the value for the PATH keyword.	#S_218DIR	The path entered on the screen.
Type in the name of a working list if this command uses one.	Press ENTER	No working list is required by this command
Do you wish to enter another JSM command?	Y	
Type in the JSM command you want to build.	LIST	
How many keywords do you want to specify?	1	
Type in the keyword name.	PATH	
Type in the value for the PATH keyword.	Blank out the field holding the previous value entered and Press ENTER	This keyword does not require a value.
Type in the name of a working list if this command uses one.	WL_218	The working list to get the list of files back.
Do you wish to enter another JSM command?	Y	
Type in the JSM command you want to build.	QUIT	

How many keywords do you want to specify?	0	This command has no keywords.
Do you wish to enter another JSM command?	N	

d. Once the BBJSMCMD template has completed, you will have all of the required code for working with the FTP service.

Your RDMLX code might appear as:

```
* BUILD THE JSM COMMAND
#JSMXCMD := CONNECT
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD HOST
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXC
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXCMD)
*
#JSMXCMD := LOGIN
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD USER
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD PASSV
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXC
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXCMD)
*
#JSMXCMD := CHGDIR
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD PATH ;
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXC
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXCMD)
*
#JSMXCMD := LIST
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD PATH ;
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXC
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXCMD)
*
#JSMXCMD := QUIT
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXC
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXCMD)
```

Your complete RDML/RDMLX code might appear as follows:

iiiFN04:

```
* =====
* Copyright .....: (C) LANSA, 1989
* Process .....: IIIPRO02
* Function .....: IIIFN04
* Created by ....: PCXUSER
* Created on ....: 08/06/23 at 10:58:57
* Description ...: Display FTP Service
* =====
* Working fields, lists and groups
* =====
DEFINE FIELD(#W_TYPE) TYPE(*CHAR) LENGTH(1) DESC('Type of file')
DEFINE FIELD(#W_FILE) TYPE(*CHAR) LENGTH(80) COLHDG('File Name')

DEF_LIST NAME(#WL_218) FIELDS(#W_TYPE #W_FILE) TYPE(*WORD)
DEF_LIST NAME(#BL_218) FIELDS(#W_FILE) ENTRYS(500)

* Group for all input fields
DEFINE FIELD(#S_218DIR) TYPE(*CHAR) LENGTH(200) DESC('FTP Directory')
GROUP_BY NAME(#GB_PNLDTA) FIELDS(#S_218HOST #S_218DIR #S_218DIR)
*
* =====
* Program Mainline : IIIFN04
* =====
*
FUNCTION OPTIONS(*DIRECT)

BEGIN_LOOP
REQUEST FIELDS(#GB_PNLDTA) BROWSELIST(#BL_218)
CLR_LIST NAMED(#WL_218)
CLR_LIST NAMED(#BL_218)
EXCHANGE FIELDS(#GB_PNLDTA)
CALL PROCESS(*DIRECT) FUNCTION(#IIIFN03) PASS_LST(#WL_218)
SELECTLIST NAMED(#WL_218)
ADD_ENTRY TO_LIST(#BL_218)
ENDSELECT
END_LOOP
```

iiiFN03:

```
* =====
* Process .....: JMIPRO02
* Function .....: JMIFN03
* Created on ....: 08/11/13 at 15:11:06
* Description ....: Use FTP Service
* Template.....: JSMXSKEL
* =====
FUNCTION OPTIONS(*DIRECT) RCV_LIST(#wl_218)
DEFINE FIELD(#W_TYPE) TYPE(*CHAR) LENGTH(1) DESC('Type of
file')
DEFINE FIELD(#W_FILE) TYPE(*CHAR) LENGTH(80) COLHDG('File
Name')
DEF_LIST NAME(#WL_218) FIELDS(#W_TYPE #W_FILE)
TYPE(*WORKING) ENTRYS(500)
*
* OPEN JSM AND VERIFY STATUS

USE BUILTIN(JSMX_OPEN) TO_GET(#JSMSTS #JSMMSG
#JSMXHDLE1)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXHDLE1)
*
* BUILD THE SERVICE LOAD COMMAND
#JSMXCMD := 'SERVICE_LOAD'
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(#JSMXCMD
'SERVICE' 'FTPSERVICE')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXHDLE1)
*
* YOUR OWN LOGIC HERE
*
* BUILD THE JSM COMMAND
#JSMXCMD := CONNECT
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(#JSMXCMD HOST
#S_218HOST)
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG)
```

```

EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXCMD)
*
#JSMXCMD := LOGIN
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(#JSMXCMD USER
#S_218USER)
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(#JSMXCMD
PASSWORD #S_218PSWD)
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXCMD)
*
#JSMXCMD := CHGDIR
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(#JSMXCMD PATH
#S_218DIR)
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXCMD)
*
#JSMXCMD := LIST
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(#JSMXCMD PATH
*BLANKS)
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG #WL_218)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXCMD)
*
#JSMXCMD := QUIT
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXCMD)
*
* UNLOAD SERVICE
#JSMXCMD := 'SERVICE_UNLOAD'
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXHDLE1)
* CLOSE JSM AND VERIFY STATUS
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1)
TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXHDLE1)

```

*

RETURN

*

* Subroutine to build JSM commands. existing JSM command

*

SUBROUTINE NAME(KEYWRD) PARMs((#W_CMDX *BOTH)
(#W_KEYWRD *RECEIVED) (#W_KEYVAL *RECEIVED))

DEFINE FIELD(#W_CMDX) REFFLD(#JSMXCMD)

DEFINE FIELD(#W_KEYWRD) REFFLD(#STD_TEXT)

DEFINE FIELD(#W_KEYVAL) REFFLD(#STD_TEXTL)

#W_CMDX += ' ' + #W_KEYWRD + '(' + #W_KEYVAL + ')'

ENDROUTINE

*

* Check the status of the JSM command issued

*

SUBROUTINE NAME(CHECK_STS) PARMs(#W_HDLE)

*

DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132)

DEFINE FIELD(#W_HDLE) TYPE(*CHAR) LENGTH(4)

*

IF COND('#JSMSTS *NE OK')

*

#MSGDTA := 'Error Status Code: ' + #JSMSTS

MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)

#MSGDTA := 'Error Message: ' + #JSMMSG

MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)

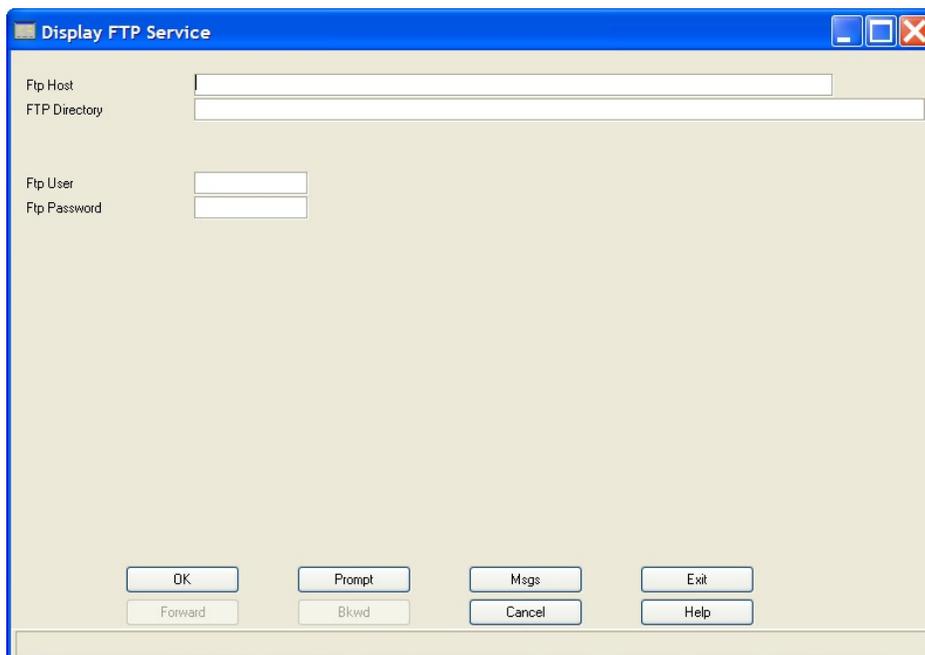
ENDIF

*

ENDROUTINE

Step 3. Test **iiiFN03** and **iiiFN04** functions

1. Compile your **iiiFN03** and **iiiFN04** functions.
2. Check that both functions compiled successfully.
3. Execute your **iiiFN04** function. Your function might appear as follows:



When the function executes, the values for the FTP Host, FTP Folder Path, FTP User and FTP Password fields are very specific to the FTP server you are using.

You should be able to obtain these details from your installation administrator if you don't know them.

FTP Host Enter the name or IP address of the FTP Server to connect to.

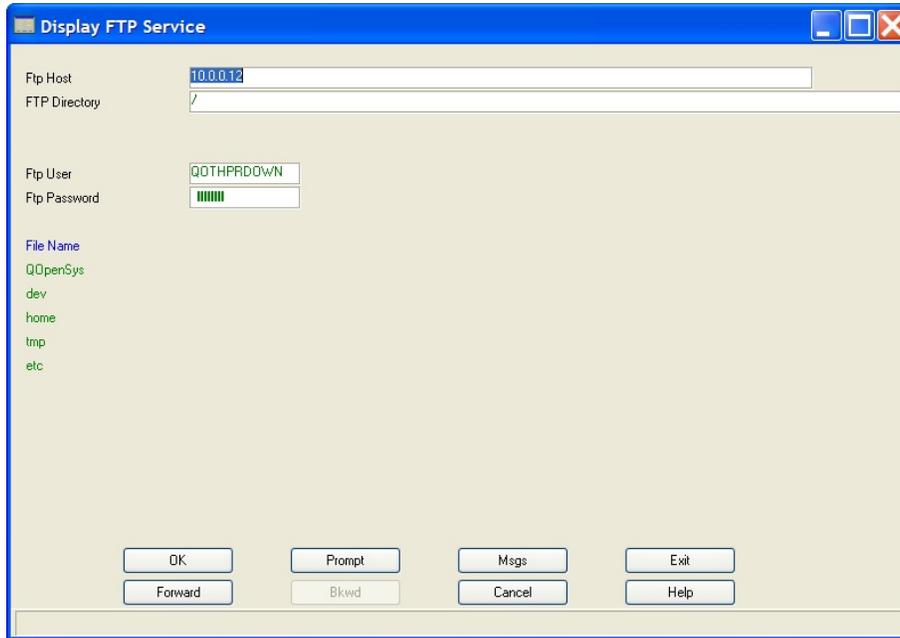
FTP Folder Path Enter the FTP folder name where to get the file list. It must be preceded by a forward slash (/). Leave blank to get the list of files from the root directory.

FTP User/FTP Password Most FTP servers require User authentication.

*** As currently defined, the password field is limited to 10 characters and upper case only. If this is unsuitable for your server, change the definition of field S_218PSWD. For example increase its length and allow lower case characters.

Once the details have been entered, press OK to get the list of files.

4. The function might appear as follows:



Step 4. Set up file JSMCLTDTA.txt - Optional

On the iSeries the information held in this file is implemented as a data area. For Windows and Linux, a flat file is used, JSMCLTDTA.txt. The file can be setup and edited using any text editor such as Notepad.

It is located in the path: c:\Program Files\LANSA\x_win95\x_lansa

The file may contain two keywords:

- DefaultServer – the default host name and port.
- ExcludePrefix – prefix for fields to be excluded from automatic exchange between JSM services and LANSA functions.

The DefaultServer setting is useful if you want to execute your functions on the desktop in Windows but access the JSM services on the iSeries.

For example the following entries specify the DefaultServer as the IP Address and Port No. 10.44.10.252:4670

localhost:4560 is the installed value for DefaultServer# If you are connecting to a different server and/or using# a different port, please change the setting below to reflect # the server and/or port, ie, # DefaultServer=MyHost:4560DefaultServer=10.44.10.252:4670Exclud

Entries beginning # are comment lines and are ignored.

See the *LANSA Integrator* guide for full details.

With the file JSMCLTDTA.txt correctly set up you can complete these tutorials by compiling and executing your functions in Windows only, without checking in to the iSeries server. When running them, your functions will access the JSM on the iSeries server.

Summary

Important Observations

- Some JSM commands pass information in a working list. Whenever a JSMX command passes a working list, you must specify the name of the working list as the third parameter of the JSMX_COMMAND. Note, when using the JSMX_COMMAND the fields to be mapped for the list is defined by the working list itself. For RDML programming, when the JSM_COMMAND is used, then a SERVICE_LIST keyword is required to define the list fields being mapped.
- It is important to understand how to use the subroutine named KEYWRD. The subroutine receives three parameters: a command, a keyword and a value.

Consider this example:

```
#S_218HOST := '10.10.10.10'  
#JSMXCMD := CONNECT  
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(#JSMXCMD HO
```

- The value JSMXCMD is mapped to W_CMDX
- The value 'HOST' is mapped to W_KEYWRD
- The value S_218HOST is mapped to W_KEYVAL
- In the KEYWRD subroutine, the JSMXCMD is concatenated with the keyword.

```
#W_CMDX += ' ' + #W_KEYWRD + '(' + #W_KEYVAL + ')'
```

- The value of JSMXCMD will now be: 'CONNECT
HOST(10.10.10.10)'.

Tips & Techniques

- The KEYWRD subroutine provides an efficient way to build the JSM commands. You will appreciate the advantages of using the KEYWRD subroutine when building more complex JSM commands in the coming exercises.

What I Should Know

- How to use the FTP service to access a FTP folder to get a list of files.

- How JSM commands are used to interact with a service.
- How to use the template BBJSMXCMD build commands for JSM services.

INT004 - Using the LANSAs User Agent

Objectives:

- To learn how to use the LANSAs User Agent to send and receive a CSV (comma separated values) file from a PC to a LANSAs function running on an IBM i or Windows Server.
- To learn how to upload data from an Excel spreadsheet to an IBM i or Windows Server.
- To create a server side RDML function that receives the CSV or Excel file sent by the LANSAs User Agent, reads the data and processes it, and sends back a response to the LANSAs User Agent.
- To show how the LANSAs User Agent can be executed in batch mode.

To achieve these objectives, you will complete the following:

Concepts

Step 1. Create a User Agent Solution and edit Host Properties file

Step 2. Create a CSV file with Column Headings to Send

Step 3. Create a Server Function iiiFN05

Step 4. JSMDirect Service Configuration

Step 5. Test iiiFN05 function

Step 6. Ignore the Column Headings

Step 7. Using a CSV without Column Headings

Step 8. Sending an Excel spreadsheet. Optional

Step 9. Invoke User Agent in Batch Mode

Summary

Concepts

The components involved in a transfer of a data file from a PC to a server via the LANSAs User Agent are:

- The data file on the PC, containing the data to be processed. This can be a CSV file, an Excel Spreadsheet or several other formats.
- The host configuration file (.lih) on the PC. This identifies where the server is, and the identifier of the server program that will handle requests.
- The PC End-User: The user selects the data file to be sent, selects the configuration file and initiates the transfer by pressing Send.
- The LANSAs User Agent: This is the application used by the end-user. It resides on the PC. It interprets the configuration file selected by the user and sends the request to start the server program. It also formats and sends the data file to JSM when requested.
- JSM: The JSM receives the name of the server program and uses the server program identifier and the host URL to locate and execute the proper function. If the Server function is running on an IBM i Server using LANSAs for IBM i, the function name entry is located in file DC@W29. If the Server function is running on a Windows Server using LANSAs, the function name entry is located in DC_W29.txt
- The Server function. This is a LANSAs function that can be on the IBM i, or on Windows. It is started by the JSM and once started it starts the HTTPService service and requests the data from the LANSAs User Agent, via JSM.

There are two examples in the *SET* guide which describe the User Agent:

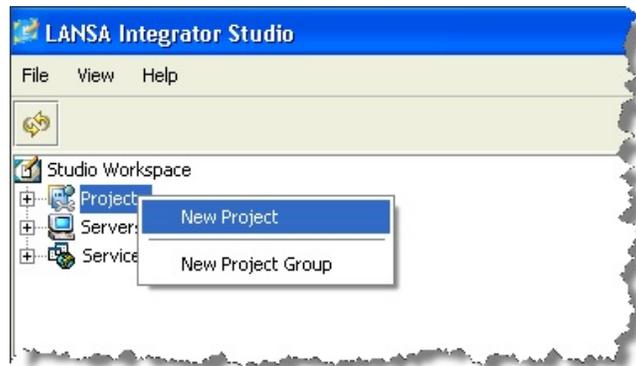
- SET 233 LANSAs User Agent - Interactive File Conversion and Upload using JSMDirect
- SET 235 LANSAs User Agent - Batch File Conversion and Upload

Note: As described above, the User Agent is small Java application which can be used by any remote location to upload a file to a server and initiate a program to process it. A User Agent solution can also be defined and run using Integrator Studio. This exercise will use Integrator Studio.

Step 1. Create a User Agent Solution and edit Host Properties file

In this step you will use Integrator Studio to define a new project *iii Training*. You will define a SendCSVFile user agent solution within this project and define the Host Properties required by this User Agent solution.

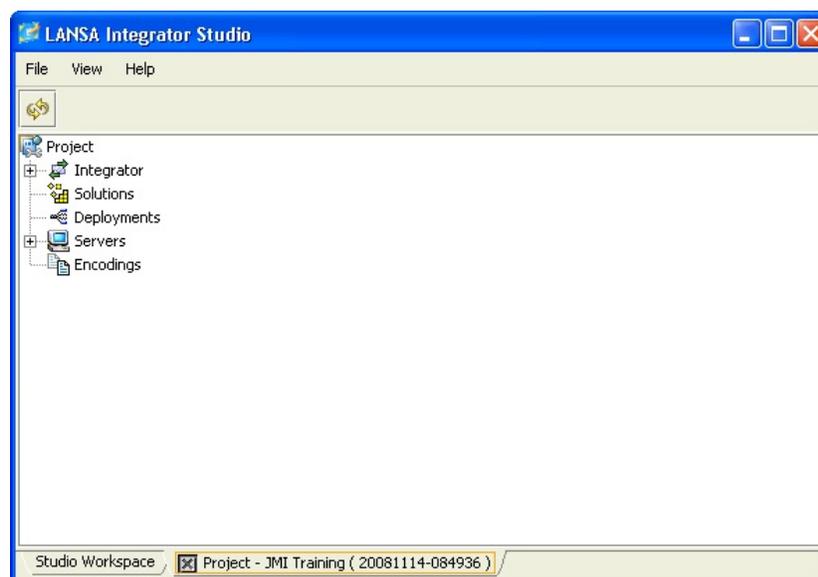
1. Open Integrator Studio. Right click on *Project*, select *New Project*.



2. Create a project called **iii Training**, where iii is your unique three letters.



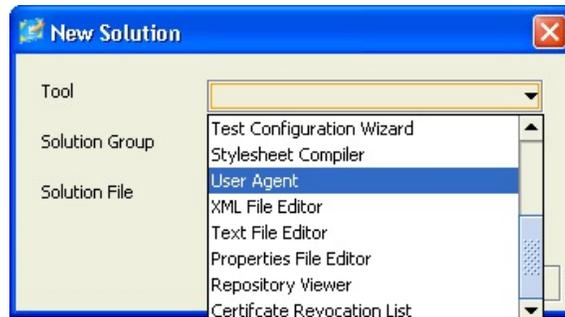
3. The project will open in a new tab:



4. Select *Solutions* and right click to create a *New Solution*:



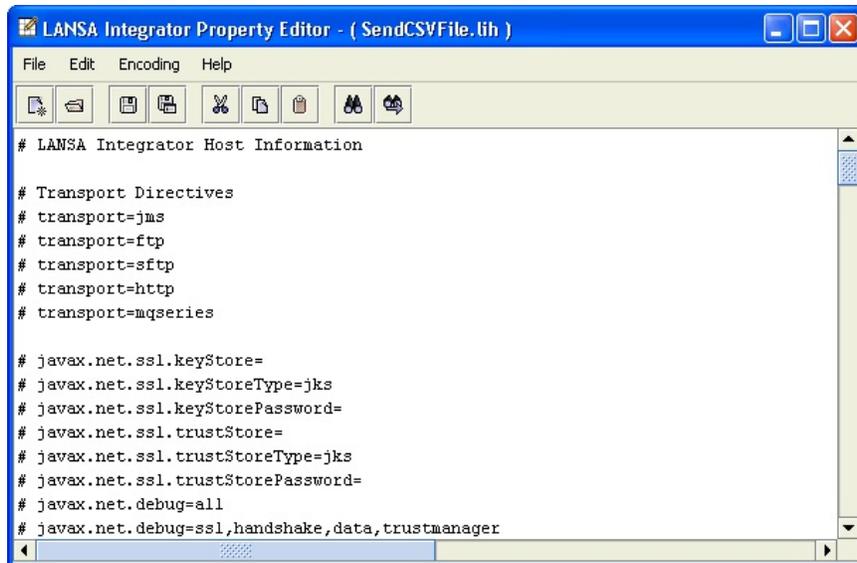
5. Select *User Agent* from the *Tool* dropdown.



6. Define a *Solution Group* and *Solution File* called **SendCSVFile** and the select **Yes** in the *Confirm* dialog.

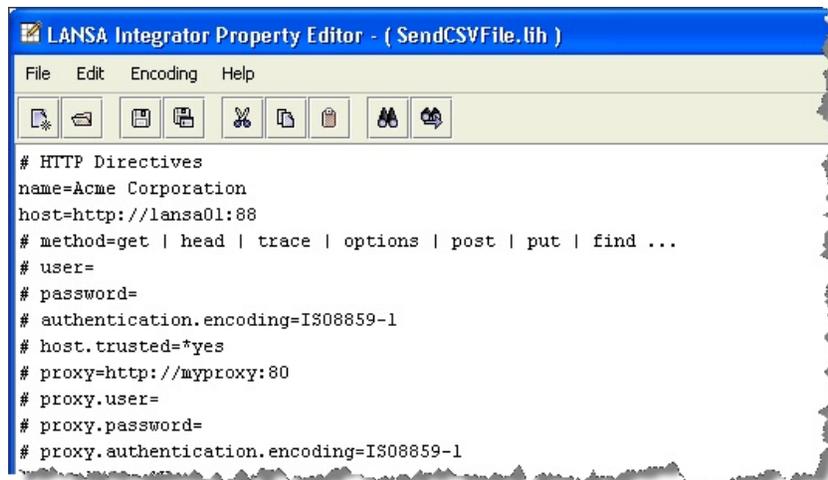


7. The *LANSA Integrator Property Editor* will open:

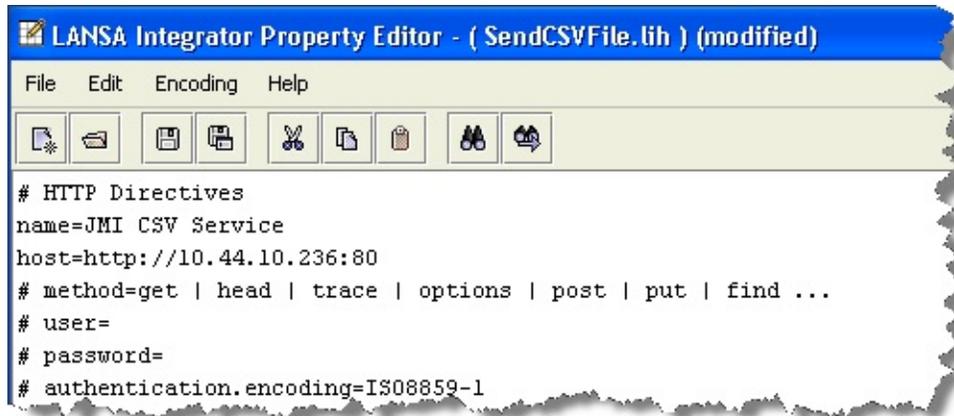


You are now going to create the *Host Information* file (.lih) which the User Agent will use. This is a configuration file which provides the parameters required by the User Agent. The User Agent will perform the upload (in this case a CSV file) based on the configuration parameters.

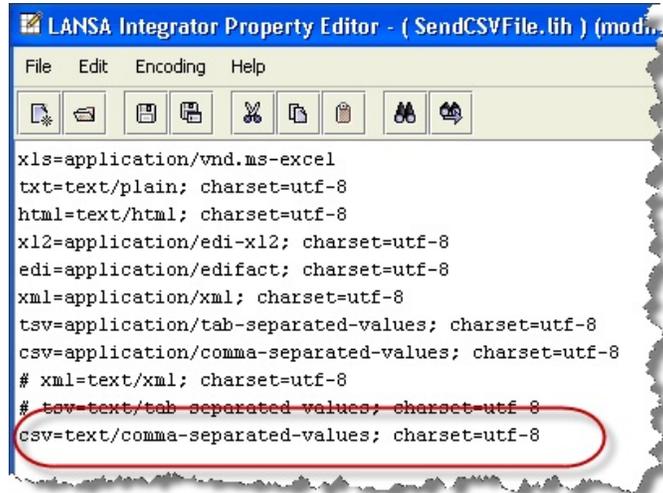
8. Scroll down to # HTTP Directives. All lines beginning with # are comments lines:



9. Change the "name" line to read name=iii CSV Service.
10. Enter the host IP address and port details that will tell the User Agent where to find JSMDirect.(If you are testing using a local Windows web server and JSM Server, you can use **http://localhost:80.**)



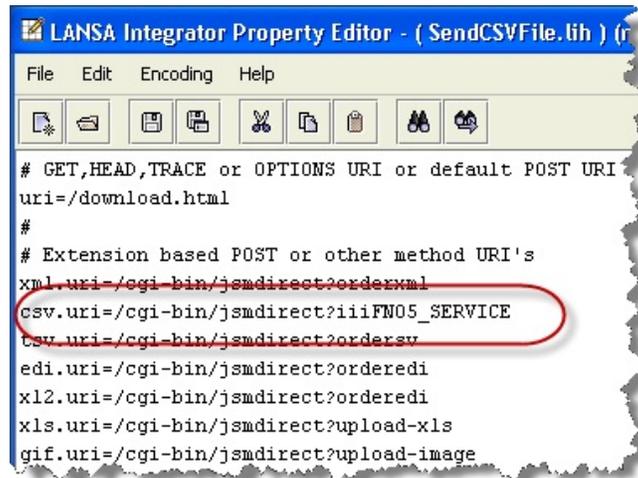
11. If your web server uses authentication, you would also enter user and password details.
12. If your web server uses a proxy, you would enter these details, including authentication details if required.
13. Enter the file type to tell the User Agent how to display a data file of this type, prior to sending it. Using the *Find* facilities in the editor, locate lines containing # CSV. Remove the # and spaces at the beginning of this line:
 csv=text/comma-separated-values; charset=utf-8



14. Again, use *Find* to locate the lines containing URI (Uniform Resource Identifier). This information will tell the User Agent that source files of type .CSV should be handled by the server program iiiFN05, which you will create in a later step.

When using a Java Services Manager (JSM) running on an IBM i server, the JSM will locate the entry in DC@W29 where the local host and server program identifier match. For a LANSA function the information recorded

includes process, function and partition. Modify the entry for csv.uri=/cgi-bin/jsmpoxy?ordersv. Your entry should look like the following:

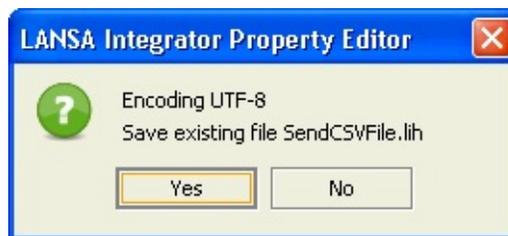


If you are using the JSM on a Windows server, the file DC_W29.txt is used to look up the function or program to call as described above. In this case your entry would look like the following:

csv.uri=/cgi-bin/jsmdirect.exe?iiiFN05_SERVICE

You will define the service entry for function iiiFN05 in [Step 4. JSMDirect Service Configuration](#).

15. Save your changes and select **Yes** in the *Confirm* dialog.



16. Exit the *Properties Editor*.

Step 2. Create a CSV file with Column Headings to Send

In this step, you will create the file containing the data to be sent using the LANSA User Agent. The file contains a list of employee numbers and the new salaries for those employees.

1. Start the *Notepad* editor in Windows.
2. Enter the following comma separated data:

```
EMPNO,SALARY,STARTDTER,DEPTMENT,SECTION  
A1004,53400,910521,ADM,01  
A1005,53500,950102,ADM,01  
A1006,53600,891201,MKT,02
```

Ensure that the employee identifiers in the EMPNO column (A1004, A1005 and A1006) exist in file PSLMST in the partition where the server side RDML function will execute. (If the data does not exist in PSLMST file, substitute employee identifiers that do exist.) Enter the rest of the data as shown.

3. Using the File menu, select the Save As option using these options:

Save locate the project directory which you created in Step 1. For example:
in C:\Program Files\LANSA\Integrator\Studio\workspace\iii
 Training\solutions\SendCSVFile\

File iii_employee_new_salary.csv
Name

Save All Files
as
Type

Step 3. Create a Server Function iiiFN05

In this step, you will write the LANSAR DMLX server function that will receive the data from the User Agent, process it, and return data to the User Agent. Your function will be very similar to the SET233S function.

1. Using the Visual LANSAR development environment, sign on to the partition nominated for the tutorials (usually DEM).
2. Create a new LANSAR process named iiiPRO03 User Agent Server Test Process, where iii is your unique 3 characters. (If the process already exists, select a different set of characters for iii.)
3. Create a new RDMLX enabled function named iiiFN05 Receive and Process Salary Amendments, belonging to process iiiPRO03, where iii is your unique 3 characters. Note that the RDMLX checkbox must be checked.
4. From the list of templates, select the template called **JSMXSKEI**.
5. Answer the template questions as shown in the table below.

Question	Answer	Comments
Do you wish to load a JSM Service?	HTTPSERVICE	

6. Edit the RDMLX code of function iiiFN05.

This function will require two working lists:

- Working list S_233RCV is used to receive the data from the CSV file sent by the User Agent. In this function, you are only interested in using the employee number and the salary even though the CSV file contains more information.
- Working list S_233SND is used to return the response from this server function to the User Agent. This list must be a different list from the receive list, because no updating of the receive list is allowed. In this case, the data returned to the User agent will be the employee number, some employee details, the new salary and a text field with a message indicating success or failure of the update.
- You will also need to define a field S_233ERRO for handling errors.

The RDMLX code might appear as follows:

```
DEF_LIST NAME(#S_233RCV) FIELDS(#EMPNO #SALARY) TYPE(*WC  
DEFINE FIELD(#S_233ERRO) TYPE(*CHAR) LENGTH(132) INPUT_A1  
DEF_LIST NAME(#S_233SND) FIELDS(#EMPNO #GIVENAME #SURN/
```

7. Modify the SERVICE_LOAD command. Use the KEYWRD subroutine to add the following keywords to the SERVICE_LOAD command

Keyword	Value
SERVICE_CONTENT	'*HTTP'
TRACE	'*YES'

The SERVICE_COMMAND(*HTTP) is required to receive HTTP posted content .

The TRACE(*YES) may be useful for testing purposes. You should remove trace before deploying to a production system.

Your completed code should look like the following. Changes are shown in red.

```
* BUILD THE SERVICE LOAD COMMAND  
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD  
'SERVICE' 'HTTPSERVICE')  
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD  
'SERVICE_CONTENT' '*HTTP')  
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD  
'TRACE' '*YES')  
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDL1  
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG)  
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXHDL1)
```

8. Locate the comment around the middle of the RDMLX source that says YOUR OWN LOGIC HERE.

On the *Design* ribbon, click on the *Templates* button and use the BBJSMXCMD template to write the RDMLX to RECEIVE the list of

employee numbers and their new salaries from the CSV file. When prompted to type in the name of the Service List, type S_233RCV. Use this table to answer the template questions.

Command	Keyword	Value	Working List
RECEIVE	HANDLER	ISVL	S_233RCV
	SVMODE	'*USE'	

Enclose *USE in single quotes, as shown.

For more detailed information on using the BBJSMXCMD template, refer to INT003 – Using the FTP Service, Step 1. Build the basic JSM Functions.

- ISVL is used for Inbound Separated Variable List.
- SVMODE of *USE tells the service that you want to use the column headings as the LANSA field names.

The RDMLX code might appear as follows:

```
* BUILD THE JSM COMMAND
#JSMXCMD := RECEIVE
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(#JSMXCMD
HANDLER ISVL)
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(#JSMXCMD
SVMODE '*USE')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG #S_233RCV)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXCMD)
```

9. Continue the YOUR OWN LOGIC SECTION by writing the RDML code to achieve the following functions:

- SELECTLIST all the entries from the received list S_233RCV.
- FETCH GIVENAME and SURNAME from PSLMST with key EMPNO.
- Check the I/O status.

If not okay, CHANGE the error field S_233ERRO to say that the employee was not found.

Else UPDATE SALARY in PSLMST.

- d. Check the I/O status of the UPDATE, and CHANGE the error field S_233ERRO accordingly.
- e. Add an entry to the working list S_233SND to be returned the LANSA User Agent.
- f. ENDSELECT

The code might appear as follows:

```
SELECTLIST NAMED(#S_233RCV)
CHANGE FIELD(#GIVENAME #SURNAME) TO(*NULL)
FETCH FIELDS(#SURNAME #GIVENAME) FROM_FILE(PSLMST) WITH
IF_STATUS IS_NOT(*OKAY)
#S_233ERRO := 'Unsuccessful: Employee not found'
ELSE
UPDATE FIELDS(#SALARY) IN_FILE(PSLMST) IO_ERROR(*NEXT) VA
IF_STATUS IS_NOT(*OKAY)
#S_233ERRO := 'Unsuccessful: Employee found but update failed'
ELSE
#S_233ERRO := 'Successful: Salary of employee ' + #EMPNO + 'successfully
updated'
ENDIF
ENDIF
ADD_ENTRY TO_LIST(#S_233SND)
ENDSELECT
```

10. Position below the above code, use the BBJS MXCMD template to write the RDMLX to SEND the response list back to the User Agent. The list will contain employee details and a success/failure message. When prompted to type in the name of the Service List, enter S_233SND. Use this table to answer the template questions.

Command	Keyword	Value	Working List
SEND	HANDLER	ISVL	S_233SND

- ISVL is used for Inbound Separated Variable List.

The code might appear as follows:

```
#JSMXCMD := SEND
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(HANDLER ISVL)
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG #S_233SND)
EXECUTE SUBROUTINE(CHECK_STS)
```

11.Delete the SERVICE_UNLOAD command.

Make sure that you leave the JSMX_CLOSE command in place.

Your finished code might appear as follows:

```
FUNCTION OPTIONS(*DIRECT)
DEF_LIST NAME(#S_233RCV) FIELDS(#EMPNO #SALARY)
TYPE(*WORKING)
DEFINE FIELD(#S_233ERRO) TYPE(*CHAR) LENGTH(132)
INPUT_ATR(LC)
DEF_LIST NAME(#S_233SND) FIELDS(#EMPNO #GIVENAME
#SURNAME #SALARY #S_233ERRO) TYPE(*WORKING)
*
* OPEN JSM AND VERIFY STATUS
USE BUILTIN(JSMX_OPEN) TO_GET(#JSMSTS #JSMMSG
#JSMXHDLE1)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXHDLE1)
*
* BUILD THE SERVICE LOAD COMMAND
#JSMXCMD := 'SERVICE_LOAD'
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD
'SERVICE' 'HTTPSERVICE')
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD
'SERVICE_CONTENT' '*HTTP')
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD
'TRACE' '*YES')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMs(#JSMXHDLE1)
*
* YOUR OWN LOGIC HERE
* BUILD THE JSM COMMAND
```

```

#JSMXCMD := RECEIVE
EXECUTE SUBROUTINE(KEYWRD) WITH_PARM(#JSMXCMD
HANDLER ISVL)
EXECUTE SUBROUTINE(KEYWRD) WITH_PARM(#JSMXCMD
SVMODE '*USE')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG #S_233RCV)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARM(#JSMXCMD)
*
SELECTLIST NAMED(#S_233RCV)
CHANGE FIELD(#GIVENAME #SURNAME) TO(*NULL)
FETCH FIELDS(#SURNAME #GIVENAME) FROM_FILE(PSLMST) WITH
IF_STATUS IS_NOT(*OKAY)
#S_233ERRO := 'Unsuccessful: Employee not found'
ELSE
UPDATE FIELDS(#SALARY) IN_FILE(PSLMST) IO_ERROR(*NEXT) VA
IF_STATUS IS_NOT(*OKAY)
#S_233ERRO := 'Unsuccessful: Employee found but update failed'
ELSE
#S_233ERRO := 'Successful: Salary of employee ' + #EMPNO + 'successfully
updated'
ENDIF
ENDIF
ADD_ENTRY TO_LIST(#S_233SND)
ENDSELECT
*
* BUILD THE JSM COMMAND
#JSMXCMD := SEND
EXECUTE SUBROUTINE(KEYWRD) WITH_PARM(#JSMXCMD
HANDLER ISVL)
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG #S_233SND)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARM(#JSMXCMD)
*
* CLOSE JSM AND VERIFY STATUS
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1)
TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARM(#JSMXHDLE1)
RETURN

```

```

*
* Subroutine to build JSM commands. existing JSM command
*
SUBROUTINE NAME(KEYWRD) PARMS((#W_CMDX *BOTH)
(#W_KEYWRD *RECEIVED) (#W_KEYVAL *RECEIVED))
DEFINE FIELD(#W_CMDX) REFFLD(#JSMXCMD)
DEFINE FIELD(#W_KEYWRD) REFFLD(#STD_TEXT)
DEFINE FIELD(#W_KEYVAL) REFFLD(#STD_TEXTL)
#W_CMDX += ' ' + #W_KEYWRD + '(' + #W_KEYVAL + ')'
ENDROUTINE
*
* Check the status of the JSM command issued
*
SUBROUTINE NAME(CHECK_STS) PARMS(#W_HDLE)
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132)
DEFINE FIELD(#W_HDLE) TYPE(*CHAR) LENGTH(4)
*
IF COND('#JSMSTS *NE OK')
*
#MSGDTA := 'Error Status Code: ' + #JSMSTS
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
#MSGDTA := 'Error Message: ' + #JSMMSG
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
ENDIF
*
ENDROUTINE

```

12. Compile the function.

13. If you are using a JSM Server on an IBM i server, check in and compile your process and function on the IBM i.

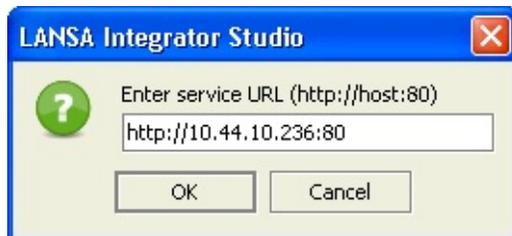
Step 4. JSMDirect Service Configuration

In this step, you will use Integrator Studio to define a *Service* and add an entry for your service and server function **iiiFN05** name.

1. Open Integrator Studio and select *Services* and use the right mouse menu to define a *New Service*.



2. Define the service **URL** and **Port**. If you are using a local Windows web server and JSM server this could be `http://localhost:80`.



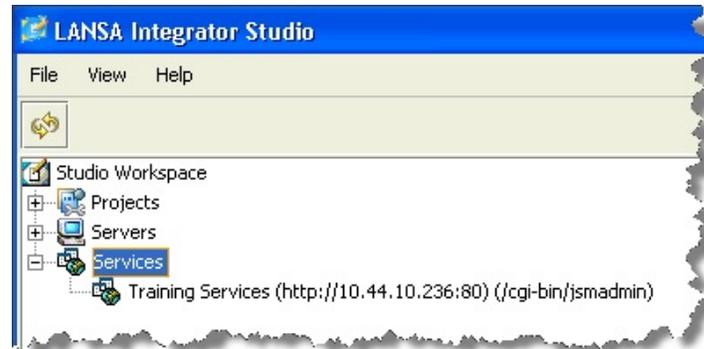
3. Select OK on the next dialog to accept the name of the JSM administration program. If you are running with a Windows Server, you will add **.exe** to **jsmadmin**



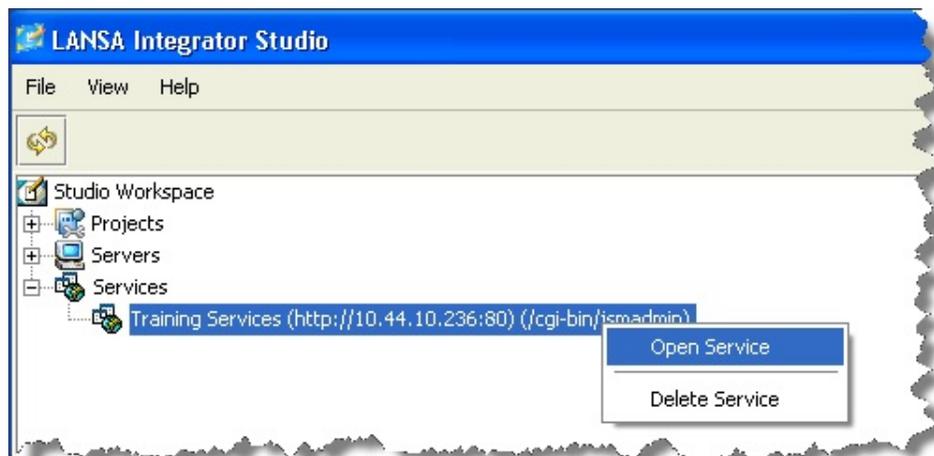
4. Enter a name for the service. This name is internal to Integrator Studio and does not need to be unique. You do not need to use your initials **iii**.



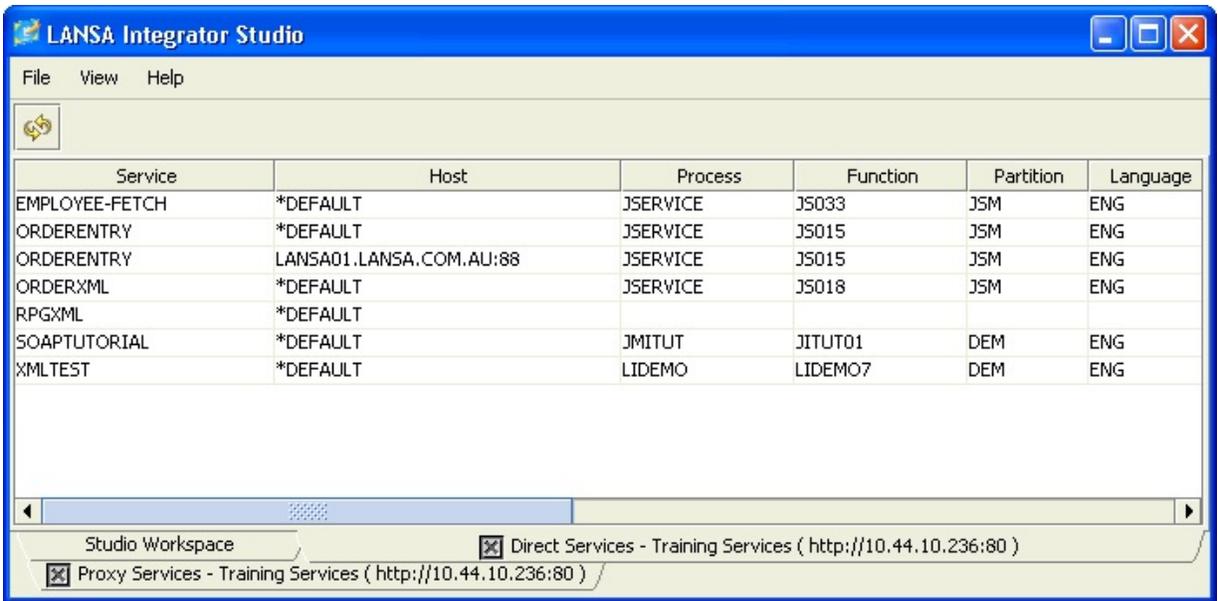
5. You are returned to the Studio workspace. Expand *Services* to see your new service definition



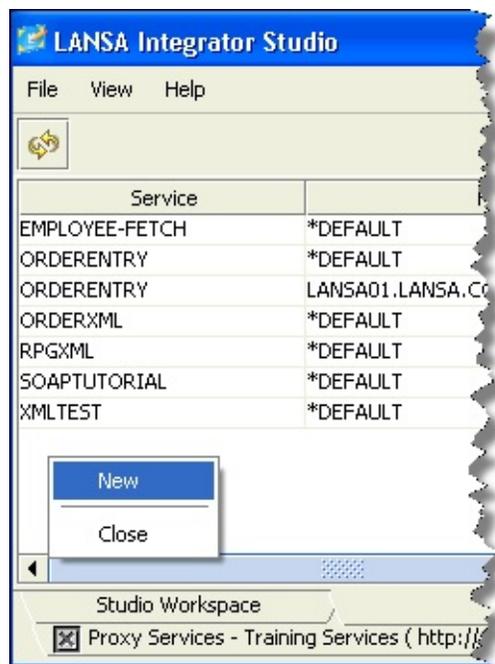
6. Select the service and using the right mouse menu select *Open Service*.



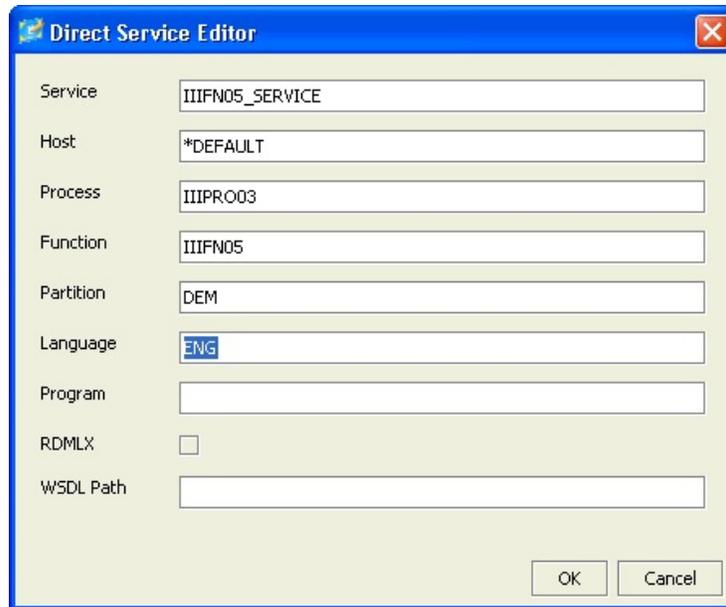
7. The entries shown will depend on what services (if any) have been defined for the JSM Server you are using.



8. Use the right mouse menu on white space and select *New*.



9. Enter suitable values for your process, function and partition as shown, and click *OK*.

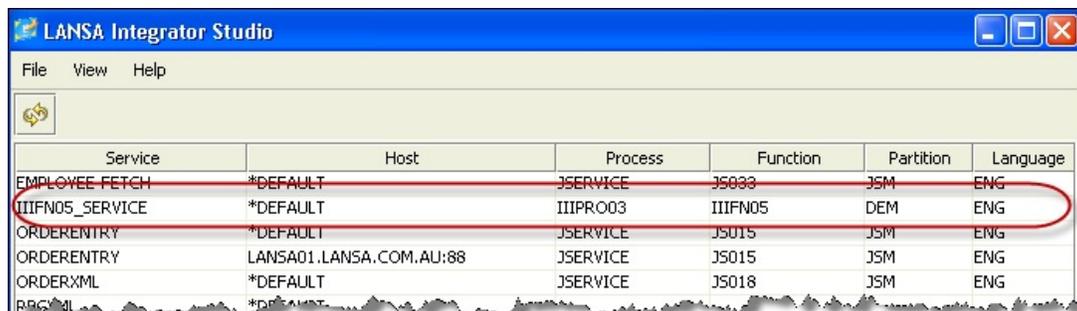


The image shows a 'Direct Service Editor' dialog box with the following fields:

- Service: IIIIFN05_SERVICE
- Host: *DEFAULT
- Process: IIIPRO03
- Function: IIIIFN05
- Partition: DEM
- Language: ENG
- Program: (empty)
- RDMLX:
- WSDL Path: (empty)

Buttons: OK, Cancel

10. You will return to the *Direct Services* tab and your new entry will be shown:



The image shows the 'Direct Services' tab in the 'LANSA Integrator Studio' application. The table below lists the services, with the newly added 'IIIIFN05_SERVICE' entry circled in red.

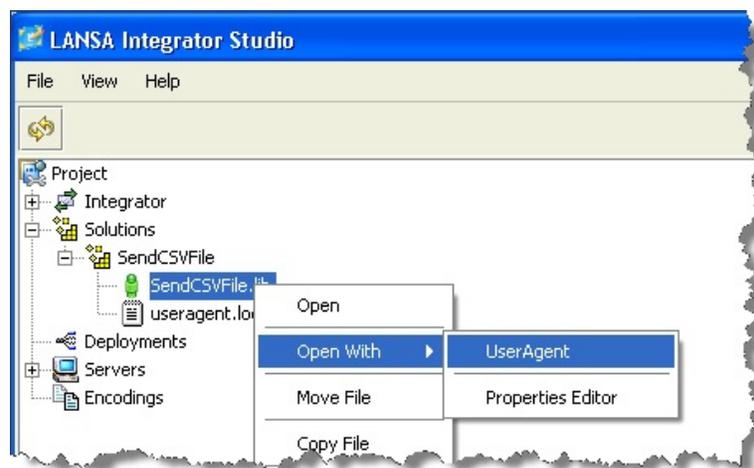
Service	Host	Process	Function	Partition	Language
EMPLOYEE_FETCH	*DEFAULT	JSERVICE	J5033	JSM	ENG
IIIIFN05_SERVICE	*DEFAULT	IIIIPRO03	IIIIFN05	DEM	ENG
ORDERENTRY	*DEFAULT	JSERVICE	J5015	JSM	ENG
ORDERENTRY	LANSA01.LANSA.COM.AU:88	JSERVICE	J5015	JSM	ENG
ORDERXML	*DEFAULT	JSERVICE	J5018	JSM	ENG
RRGXML	*DEFAULT				

11. If your service is using a Windows web server and JSM server, file DC_W29.txt has been updated. If you are using an IBM i JSM server, file DC@W29 in library DCXJSMLIB (or your equivalent) will have been update. The *User Agent* will now be able to call this service and it will resolve to a call to your function iiiFN05.

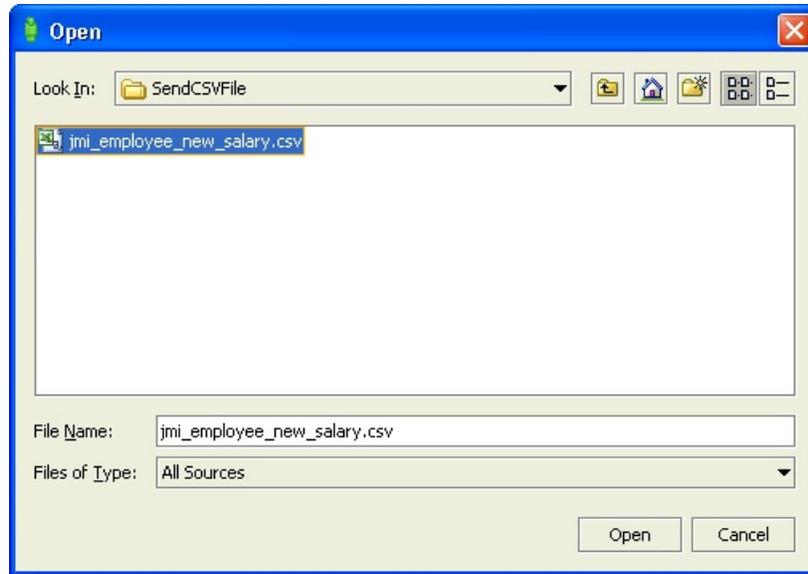
Step 5. Test iiiFN05 function

In this step, you will run the User Agent and send your data file to JSMDirect. A LANSA function will be invoked to process your data and send back a response.

1. Ensure that your Web server and JSM Server are started.
2. Open Integrator Studio if necessary and double click on *Projects/iii Training* to open your project in a new tab.
3. In the *Project* tab, open *Solutions/SendCSVFile* and use the right mouse menu to *Open With the User Agent*.



4. The User Agent has opened using the Host Configuration file (.lih) which you defined earlier (shown in the Title bar). Use the *Open* toolbar button (highlighted above) to open your CSV file.



5. The file opened in the User Agent might appear as follows:

LANSA Integrator User Agent - (SendCSVFile.tih) JMI CSV Service

File Help

1	2	3	4	5
EMPNO	SALARY	STARTDTER	DEPTMENT	SECTION
A1004	53400	910521	ADM	01
A1005	55466	950212	ADM	02
A1006	66567	921230	MKT	02

6. In the File menu, select the Send option, or use the  toolbar button.

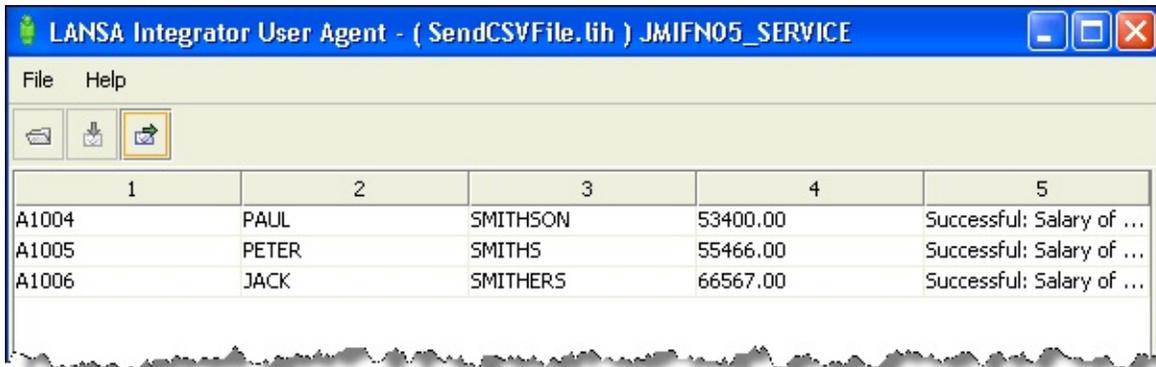
7. When the confirmation request window comes up, select Yes:



8. A completion message should display:



9. You can view the response from the server by clicking on the *Response* tab:



10.If you right mouse click on the table, you can save the response as a .CSV file.

Step 6. Ignore the Column Headings

In this step, you will modify function iiiFN05 to ignore the column headings and map the data using the fields defined in the working list S_233RCV

1. Edit the function iiiFN05.

Locate the definition of the working list used to receive the data from the User Agent. The working list is called #S_233RCV and it might appear as follows:

```
DEF_LIST NAME(#S_233RCV) FIELDS(#EMPNO #SALARY) TYPE(*WC
```

2. Modify the definition of S_233RCV so that it matches the format of the CSV file. Your code might appear as follows:

```
DEF_LIST NAME(#S_233RCV) FIELDS(#EMPNO #SALARY #STARTDTI
```

3. Locate the commands that execute the KEYWRD subroutine to build the RECEIVE command. Change the value of the SVMODE keyword to *IGNORE. One example of why you might use *IGNORE is when the column headings are not LANSA field names. For example, the server might receive a CSV file where the column headings say 'Employee Number' instead of EMPNO.

Your code might appear as follows:

```
CHANGE FIELD(#JSMXCMD) TO(RECEIVE)
EXECUTE SUBROUTINE(KEYWRD) WITH_PARM(#JSMXCMD
HANDLER ISVL)
EXECUTE SUBROUTINE(KEYWRD) WITH_PARM(#JSMXCMD
SVMODE '*IGNORE')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDL1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG #S_233RCV)
EXECUTE SUBROUTINE(CHECK_STS)
```

4. Compile the function.
5. Use the User Agent to test the function as before. The result should be exactly the same.
6. **Optional.** Edit your CSV file to contain column headings such as "Number, Annual Salary, Start Date, Department, Section Code" and save your

changes. If the User Agent is still open, use *File / Close Source* and then open your modified CSV file and send it. Once again, you should get the same results.

Step 7. Using a CSV without Column Headings

In this step, you will modify the CSV file to remove the column headings. The RDMLX will need to be modified to change the SVMODE keyword on the RECEIVE command to *NONE. i.e. the CSV file has no column headings. In Step 6, you modified the received working list definition to match the columns in the CSV file, so no other changes are needed to receive the CSV file. In fact, since your function only requires the first two fields, the working list could be defined as containing EMPNO and SALARY only.

You will add a SVHEAD keyword to the SEND command with a value of iiiFN05. This will look up a project entry you will add to the HTTPService properties file for the header row to be added to the sent CSV file.

1. Start the Notepad editor in Windows to open the file **iii_employee_new_salary.csv** created at the beginning of this exercise. The file might appear as follows:

```
EMPNO,SALARY,STARTDTER,DEPTMENT,SECTION
A1004,53400,910521,ADM,01
A1005,53500,950102,ADM,01
A1006,53600,891201,MKT,02
```

2. Remove the first row in the file that contains the column heading data. The file might appear as follows:

```
A1004,53400,910521,ADM,01
A1005,53500,950102,ADM,01
A1006,53600,891201,MKT,02
```

3. Using the File menu, select the Save to save the file with the modified contents. You can close the file **iii_employee_new_salary.csv**.
4. Edit the function **iiiFN05**.

Locate the definition of the working list used to receive the data from the User Agent. The working list is called **#S_233RCV** and it might appear as follows:

```
DEF_LIST NAME(#S_233RCV) FIELDS(#EMPNO #SALARY #STARTDTI
```

5. Locate the commands that execute the KEYWRD subroutine to build the RECEIVE command.

Change the value of the SVMODE keyword to *NONE. SVMODE(*NONE) is the default value so you could simply omit the keyword altogether.

The RDMLX might appear as follows:

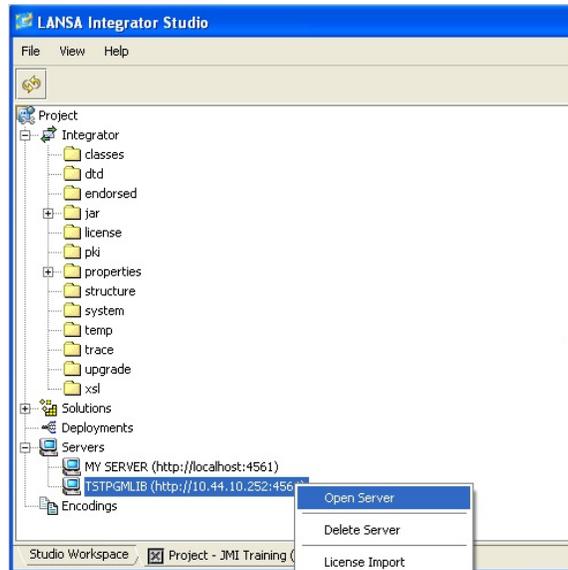
```
CHANGE FIELD(#JSMXCMD) TO(RECEIVE)
EXECUTE SUBROUTINE(KEYWRD) WITH_PARM(#JSMXCMD
HANDLER ISVL)
EXECUTE SUBROUTINE(KEYWRD) WITH_PARM(#JSMXCMD
SVMODE '*NONE')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMCMD) TO_GET(#JSMSTS #JSMMSG #S_233RCV)
EXECUTE SUBROUTINE(CHECK_STS)
```

6. Locate the execute KEYWRD subroutine code for the SEND command. Add a SVHEAD keyword with a value of iiiFN05. Your code should look like the following:

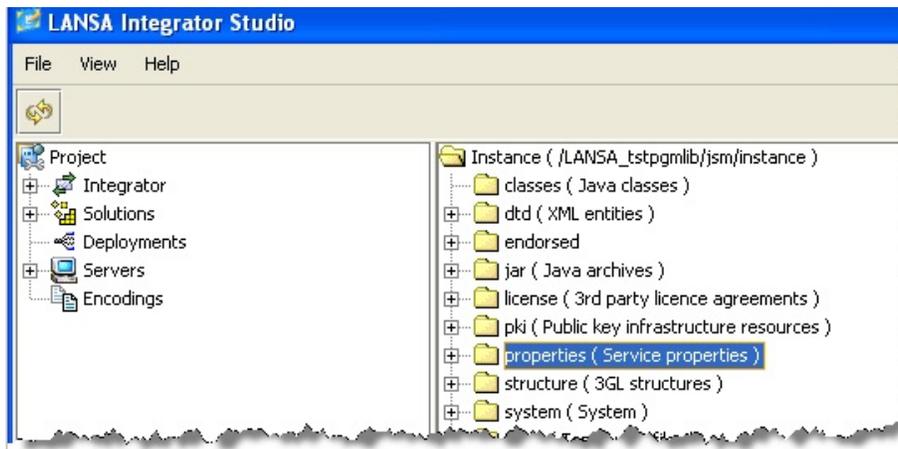
```
* BUILD THE JSM COMMAND
#JSMXCMD := SEND
EXECUTE SUBROUTINE(KEYWRD) WITH_PARM(#JSMXCMD
HANDLER ISVL)
EXECUTE SUBROUTINE(KEYWRD) WITH_PARM(#JSMXCMD
SVHEAD iiiFN05)
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMSTS #JSMMSG #S_233SND)
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARM(#JSMXCMD)
```

7. Compile the function.
8. In this step you will add a section to the HTTPSERVICE properties file for your Project. This section will define the column headings to be used for the sent CSV file.
 - a. Open your **iii Training** project in Integrator Studio.

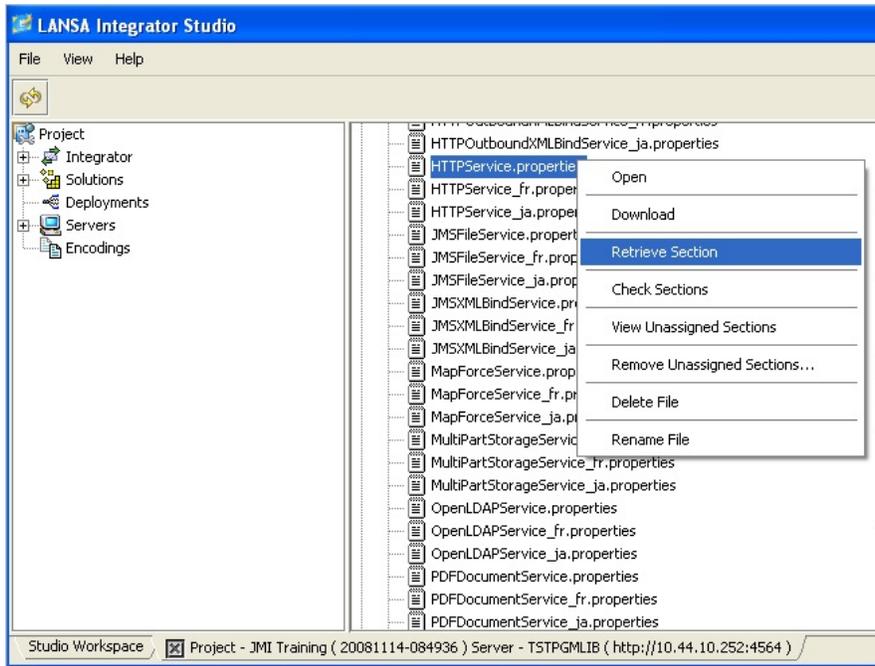
On the *Project* tab, select the *Server* you are using for training and use the right mouse menu to *Open Server*.



b. The server instance will open on the right hand side:



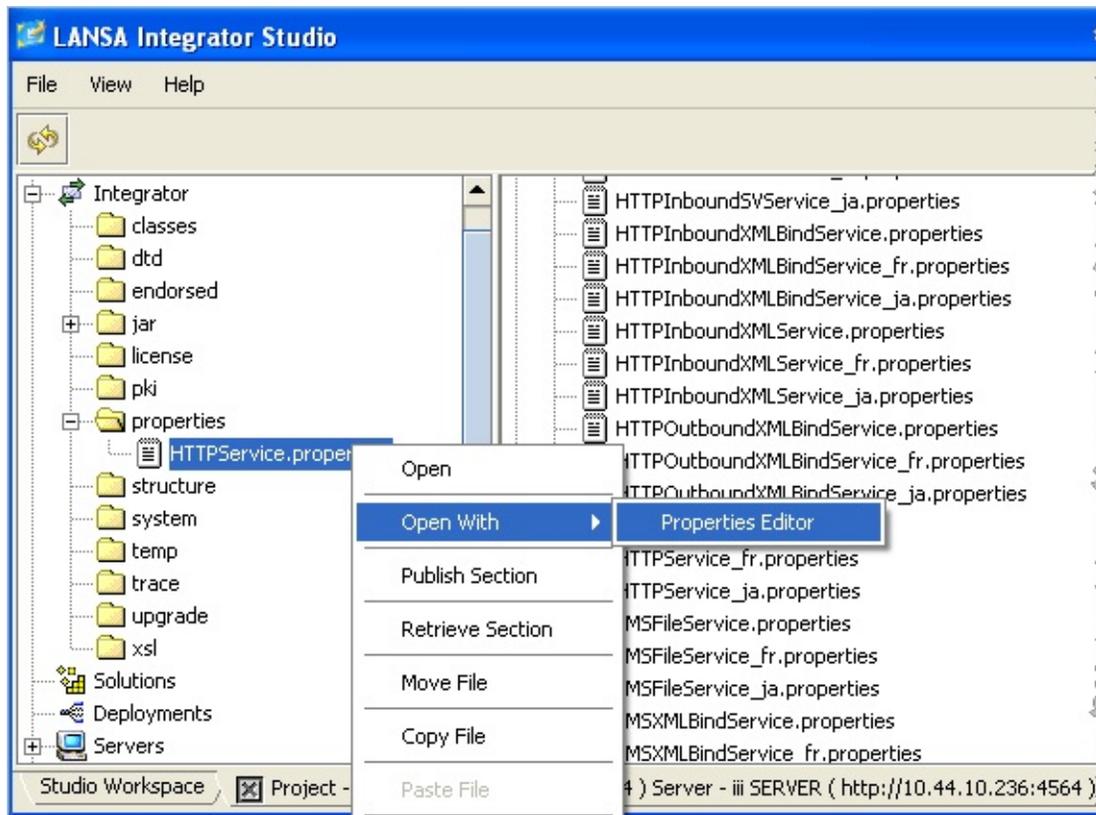
c. Open the *properties* level, and scroll down to `HTTPService.properties`. Select this and use the right mouse menu to *Retrieve Section*.



- d. The following dialog box will be shown, since you do not have an HTTPService.properties section defined for your project at this point. Select *OK* and an entry HTTPService.properties will be created under *Integrator / Properties* on the left hand side. i.e. in your project.



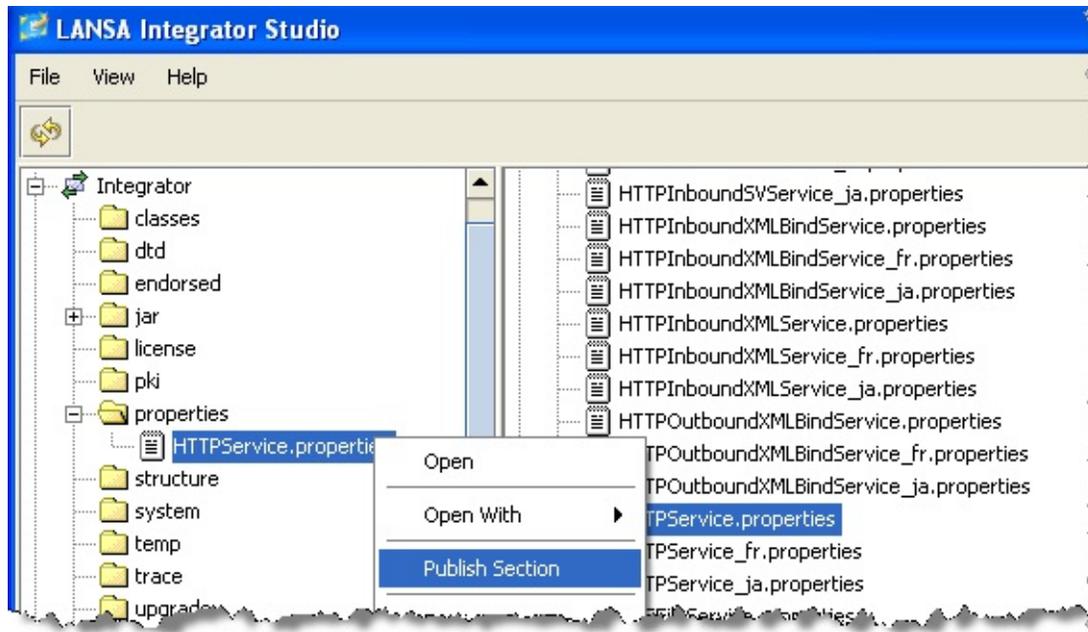
- e. Select *Integrator / Properties / HTTPService.properties* and use the right mouse menu to *Open With / Properties Editor*.



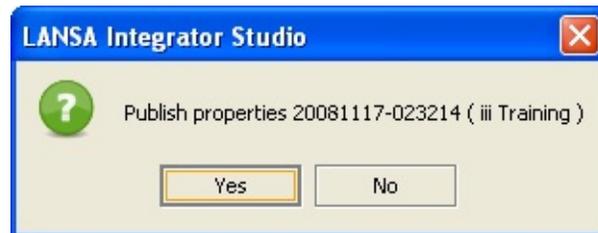
- f. The *Properties Editor* will open. Add an entry which defines column headings for separated variables files, as shown in the screen shot. Save your changes and close the properties editor.



- g. Select `HTTPService.properties` in your project and use the right mouse menu to select *Publish Section*.



- h. Select **Yes** in the *Confirm* dialog. This will add a section at the end of the `HTTPService.properties` file for your project.



- i. On the right hand side (the JSM server instance) use the right mouse menu to select *Properties / HTTPService.properties* to *Open* the JSM server `HTTPService` properties file. Scroll to the end to view the entry you have just added.

```

# Move these entries to a new or existing studio-project section.
# Delete this unassigned section.
#
#
#!</studio-project>
#
#
#!<studio-project id="20081114-084936" name="JMI Training">
sv.head.JMIFN05=EMPNO,SALARY,STARTDTER,DEPARTMENT,SECTION
sv.head.JMIFN050=EMPNO,GIVENAME,SURNAME,SALARY,ERRORS
#!</studio-project>
#
#
#!<studio-project id="20081117-023214" name="iii Training">
sv.head.iiiFN05=EMPNO,GIVENAME,SURNAME,SALARY,ERRORS
#!</studio-project>
#
#

```

8. Use the instructions in the previous step to test the function. The response should look like the following:

1	2	3	4	5
EMPNO	GIVENAME	SURNAME	SALARY	ERRORS
A1004	PAUL	SMITHSON	53400.00	Successful: Salary of ...
A1005	PETER	SMITHS	55466.00	Successful: Salary of ...
A1006	JACK	SMITHERS	66567.00	Successful: Salary of ...

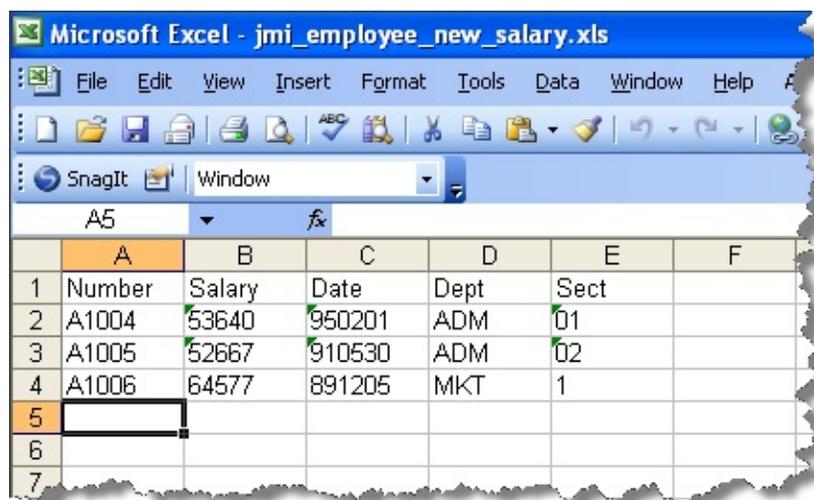
Step 8. Sending an Excel spreadsheet. Optional

Microsoft Excel must be installed on your PC in order to complete this step.

In this step, you will create an Excel spreadsheet and modify the host configuration file SendCVSFile.lih to allow the User Agent to use an .xls file as source, instead of a .csv file. The User Agent will read the spreadsheet and display the contents. It will then send the data to the Server in CSV format for processing by function iiiFN05. Function iiiFN05 will return a response to the LANSA User Agent. Function iiiFN05 must have a SVMODE(*IGNORE) keyword on the RECEIVE command, since the Excel spreadsheet will be defined with column headings which are not field names. If necessary you should change function iiiFN05.

First, you must create the file containing the data to be sent using the User Agent. The file contains a list of employee numbers and the new salaries for those employees.

1. Start Excel in Windows.
2. Enter the following data in the top left 15 cells of sheet 1, as shown:



The screenshot shows a Microsoft Excel window titled "Microsoft Excel - jmi_employee_new_salary.xls". The spreadsheet has the following data:

	A	B	C	D	E	F
1	Number	Salary	Date	Dept	Sect	
2	A1004	53640	950201	ADM	01	
3	A1005	52667	910530	ADM	02	
4	A1006	64577	891205	MKT	1	
5						
6						
7						

Important Notes:

- Format cells E1 to E3 as text prior to entering the values, otherwise Excel will treat the entered values as numbers.
- Columns headings should be included in the Excel spreadsheet. Your function assumes headings exist and ignores them. **Note:** this also means

that without a column headings row, your function would ignore the first row of data.

Ensure that the employee identifiers in the EMPNO column (A1004, A1005 and A1006) exist in file PSLMST in the partition where the server side RDML function will execute. (If the data does not exist in PSLMST file, substitute employee identifiers that do exist.) Enter the remaining data as shown.

3. In the File menu, select the Save As option using these options:

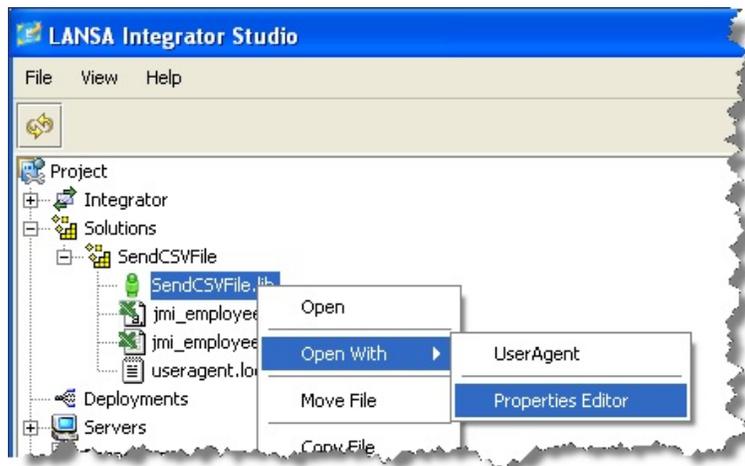
Save locate the project directory where you defined the send CSV file
in solution. For example: c:\program
 files\lansa\integrator\Studio\workspace\iii
 Training\solutions\SendCSVFile\

File iii_employee_new_salary.xls
Name

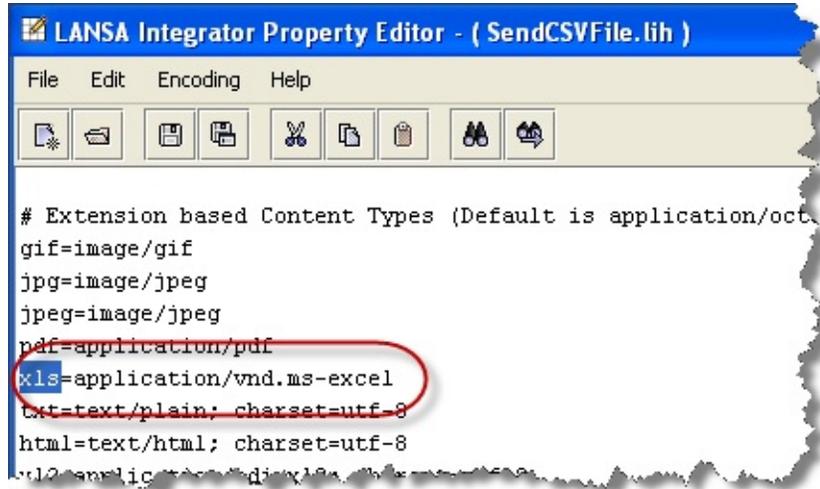
Save Microsoft Excel workbook (*.xls)
as
Type

4. Close Excel.

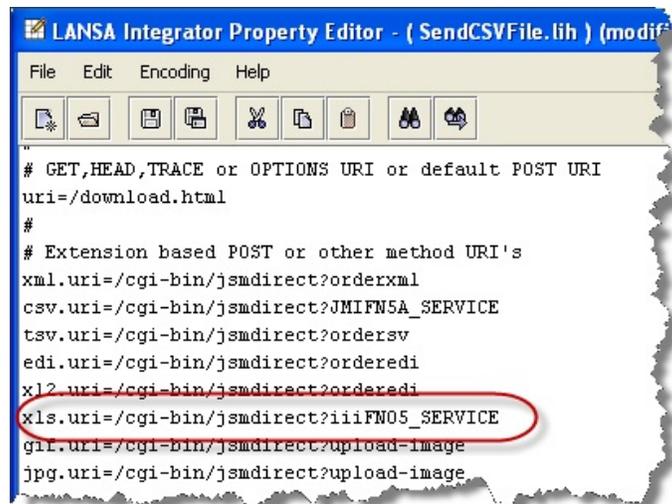
5. With your **iii Training** project open in Integrator Studio, open the SendCSVFile.lih with the *Properties Editor*.



6. Use *Find* to locate xls entries. You should find the highlighted entry already exists. This line defines the Windows application which the User Agent will use to display an .XLS file.



7. Locate entries for **uri** and modify the entry for **xls.uri=/cgi-bin . . .** as shown in the screen shot. This defines the service which the JSM will look up in DC@W29 to find the program or function to call when handling an xls file.



If you are using the JSM server on Windows the entry would change to:

xls.uri=/cgi-bin/jsmdirect.exe?iiiFN05_SERVICE

8. Add a new section to the end of the configuration file, to tell the LANSA User Agent which of the sheets in the Excel spreadsheet to read and what to convert the spreadsheet to, prior to display.

#

```
# Excel processing
#
excel.convert=text/comma-separated-values;charset=utf-8
excel.sheet=Sheet1
```

9. Save your changes and close the *Properties Editor*
10. As before, open the User Agent for your SendCSVFile solution, open the Excel file `iii_employee_salary_amendments.xls` and send it to the JSM server. The response should be the same as before.

Step 9. Invoke User Agent in Batch Mode

The previous steps in this exercise have run the User Agent from Integrator Studio. In this step, you will execute the User Agent in batch mode to send the XLS file. **Note:** this step requires the User Agent to be installed as a standalone application.

1. The files already created for the User Agent project are in a directory such as:

C:\Program Files\LANSA\Integrator\Studio\workspace\JMI
Training\solutions\SendCSVFile

Copy these files from the above directory

SendCSVFile.lih

iii_employee_new_salary.xls

to the \User Agent\workspace folder, such as:

C:\Program Files\LANSA\Integrator\UserAgent\workspace

2. In the following steps you will use *Notepad* to create an iii_UPLOAD.BAT file in the directory where you installed the LANSA User Agent (by default **C:\Program Files\LANSA\Integrator\UserAgent**). The file iii_UPLOAD.BAT is a DOS batch file which will enable the user agent to be executed without a user interface, by passing the required parameters into the User Agent when the batch file is executed.

3. Open *Notepad* and open the file:

C:\Program

Files\LANSA\Integrator\UserAgent\workspace\upload.bat

Note: You will need to use *Files of Type: All Files* when opening the file.

This is a shipped example batch file for running User Agent. The code should look like the following:

```
@echo off000  
cls
```

```
rem --- Delete previous error log file
```

```
if exist useragent.err del useragent.err
```

```
rem --- Send
```

```
java -Djsf.log=\LANSA\Integrator\useragent\useragent.log
com.lansa.jsf.useragent.JSFUserAgent acme.lih order.xls xls-order-
response.rsp
if exist useragent.err goto error
```

```
rem --- Send
java -Djsf.log=\LANSA\Integrator\useragent\useragent.log
com.lansa.jsf.useragent.JSFUserAgent acme.lih order.csv csv-order-response.rsp
if exist useragent.err goto error
```

```
goto end
rem -- An error has occurred
:error
cls
echo Check useragent.err for possible messages
echo Check useragent.log for possible messages
goto end
:end
```

Delete the second block of code between "rem --- Send" and "goto error" including this code. The supplied example assumes that the batch file will send two files. The code to delete, is shown in red in the program code above. Your code should now look like the following:

```
@echo off
cls
rem --- Delete previous error log file
if exist useragent.err del useragent.err
rem --- Send
java -Djsf.log=\LANSA\Integrator\useragent\useragent.log
com.lansa.jsf.useragent.JSFUserAgent acme.lih order.xls xls-order-
response.rsp
if exist useragent.err goto error
goto end
rem -- An error has occurred
:error
cls
echo Check useragent.err for possible messages
echo Check useragent.log for possible messages
```

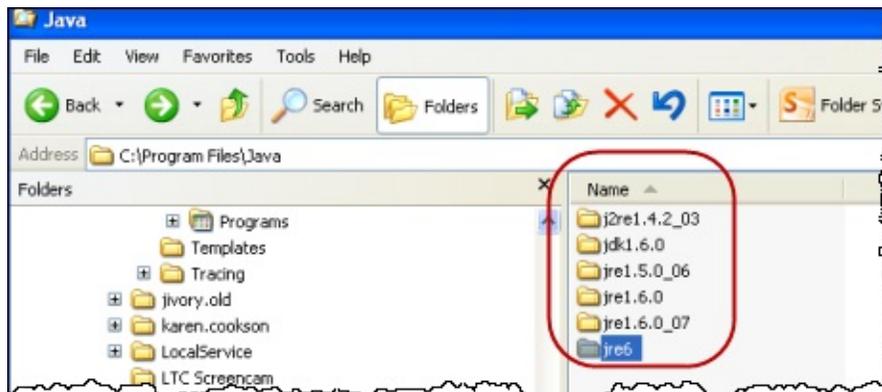
```
goto end
:end
```

- The supplied code needs to be extended to operate on your PC. The code you enter will depend on the version of Windows you are using. The examples provided here are for Windows XP. Replace "java" at the start of the command line with a full path. Your code should look like the following. The new code is shown in red. REVIEW THIS. NOT SURE OF CORRECT FORMAT.

```
rem --- Send
```

```
C:\WINDOWS\system32\JAVA.W.EXE "-Djava.ext.dirs=..\lib\ext" "-Djava.endorsed.dirs=..\lib\endorsed" "-Djsf.log=.\useragent.log" com.lansa.jsf.useragent.JSFUserAgent acme.lih order.xls xls-order-response.rsp
```

- The command line you are editing runs the User Agent in the Java runtime environment (JVM) passing a number of parameters such as the configuration file (.lih) to be used. The first parameter begins "-Djava.ext.dirs= . You need to insert into this parameter the path for the \lib\ext folder. If you examine the folder C:\Program Files\Java using Windows explorer you will probably find it contains folders for a number of versions of Java, which are left in place when Java is updated. See the example following:



You should specify the path for the latest version of Java (or alternatively the actual Java version you have specified in your Integrator settings). For example C:\Program Files\Java\jre6\lib\ext. Add this path into the first parameter as shown. Changes are shown in red:

```
rem --- Send
```

```
C:\WINDOWS\system32\JAVA.W.EXE "-Djava.ext.dirs=c:\program
```

```
files\java\jre6\lib\ext;.lib\ext" "-Djava.endorsed.dirs=..\lib\endorsed" "-Djsf.log=.\useragent.log" com.lansa.jsf.useragent.JSFUserAgent acme.lih order.xls xls-order-response.rsp
```

6. This step will specify the correct runtime parameters for the configuration file(lih), the input file (xls) and the response file (csv).

Replace the end of the command string, starting from **acme.lih**, so that the command looks like the following. Ensure the file names used have your initials. The changes are shown in red:

```
rem --- Send
C:\WINDOWS\system32\JAVA.W.EXE "-Djava.ext.dirs=c:\program
files\java\jre6\lib\ext;.lib\ext" "-Djava.endorsed.dirs=..\lib\endorsed" "-Djsf.log=.\useragent.log" com.lansa.jsf.useragent.JSFUserAgent
workspace\SendCSVFile.lih workspace\iii_employee_new_salary.xls
workspace\iii_employee_response.csv
```

Important Note: Remember you are editing a single command line. It is shown here spilt across a number of lines due to width limitations. Using *Notepad* you can view this line as a single line or over a number of lines if word wrap is enabled.

7. The complete iii_upload.bat file should look like the following:

```
@echo off
cls
```

```
rem --- Delete previous error log file
if exist useragent.err del useragent.err
```

```
rem --- Send
C:\WINDOWS\system32\JAVA.W.EXE "-Djava.ext.dirs=C:\Program
files\Java\jre1.6.0_07\lib\ext;.lib\ext" "-Djava.endorsed.dirs=..\lib\endorsed"
"-Djsf.log=.\useragent.log" com.lansa.jsf.useragent.JSFUserAgent
workspace\SendCSVFile.lih workspace\iii_employee_new_salary.xls
workspace\iii_employee_response.csv
if exist useragent.err goto error
```

```
goto end
```

```
rem -- An error has occurred
```

```
:error
cls
echo Check useragent.err for possible messages
echo Check useragent.log for possible messages

goto end

:end
```

Note how the send instruction is composed of four parts:

- The instruction to start the User Agent.
- The first parameter - SendCSVFile.lih (how to find the service)
- The second parameter - iii_Employee_New_Salary.xls (the data to be processed)
- The third parameter - iii_employee_response.csv (the name of the file that will receive the response from the service)

8. In the File menu, select the Save As option using these options:

Save locate the directory where you installed the LANSAs User Agent, by
in default ...\\program files\\lansa\\integrator\\UserAgent\\

File iii_UPLOAD.BAT

Name

Save All files

as

Type

Note: Save to ...\\UserAgent\\ not ...\\UserAgent\\workspace\\.

9. Execute the **iii_UPLOAD.BAT** file. If successful, a file called **iii_response.csv** should be created in the directory ...\\UserAgent\\workspace\\. Windows will open this (CSV) file with Excel.

If you encounter a problem, the first thing to check is whether you can manually start the LANSAs User Agent, load host SendCSVFile.lih, open

source `iii_Employee_New_Salary.xls`, send, and receive a response.
Refer to [Step 5. Test iiiFN05 function](#) for details.

Summary

Important Observations

- The Server program identifier in the configuration file is a logical value that JSM resolves to the server side process/function, partition and language by looking up the file DC@W29.
- The SVMODE should be *NONE when the CSV file contains no column heading information. When the SVMODE is *NONE you have the option of using the SVHEAD keyword to specify the CSV file format in HTTPService.properties file. If you do not use an SVHEAD, the format of the working list to receive the data must match that of the CSV file.
- The SVMODE could be *USE or *IGNORE when the CSV file contains column heading information. For the mode *USE, the column headings in the CSV file must be LANSA field names and the format of the working list is not required to match that of the CSV file.
- For the mode *IGNORE, the column headings in the CSV file can be anything and the format of the working list must match that of the CSV file.
- It is possible to automate the user actions into a single .bat file that when run will start the LANSA User Agent, specify the configuration file, specify the data file to be processed, send, and specify the name of the local file to receive the response from the server.
- It is possible to configure the LANSA User agent to take Excel spreadsheets and other formats as data. These will be converted to CSV before being sent. No change is required to the Server side function to handle these formats.

Tips & Techniques

- The format of the CSV data determines the SVMODE used to RECEIVE the file on the server.
- If you are testing using a Windows Web server and JSM server install on your own PC, you can use a host address of host=http://localhost:80 when specifying the host=http://nnn.nnn.nnn.nnn:pp.
- The LANSA User Agent also supports the saving of data to an Excel spreadsheet.

What I Should Know

- How to use the User Agent to send and receive data to a function running on an IBM i.

- How to use the JSM commands SEND and RECEIVE in Server Side functions using the HTTP Client service
- When to use the different modes to receive the files on the Server side.

INT005 - Department Inquiry Bindings

Objectives:

- To learn how to handle XML files with the LANSAs Integrator.
- To learn how to use Integrator Studio to create projects and manage files on the JSM Server.
- To use the XML Binding Wizard to create XML binding jar files.
- To create bindings to receive a Department Code to a function and then send back a Department Description (Server).
- These bindings will be used in the INT006 exercise, which creates the server functions.

To achieve these objectives, you will complete the following:

- [Concepts](#)
- [Step 1. Extend Studio Project iii Training](#)
- [Step 2. Create Request XML - iiiPRO04_request.xml](#)
- [Step 3. Create Response XML - iiiPRO04_response.xml](#)
- [Step 4. Create iiiFN06 Server Inbound \(request\) binding](#)
- [Step 5. Create iiiFN06 Server Outbound \(response\) binding](#)
- [Step 6. Binding Deployment and Configuration](#)
- [Summary](#)

Concepts

To begin, you will learn a few JSM specific concepts related to XML files and bindings. Writing or obtaining the XML would usually be the first step in your application. The XML is required by the *XML Binding Wizard* to map the bindings.

There will usually be two XML files involved when writing the client and server side of an application. In this course, they will be referred to, as the **request** and **response** XML:

- The **request** XML is the one sent by the client and received by the server side. You will send a request containing a Department Code from the DEPTAB table.
- The **response** XML is the one sent back by the server. It is usually different to the request XML received by the client. In this case, the response will be the Department Description from the DEPTAB table.

Using an XML file as input, the *XML Binding Wizard* can create a binding to map the XML to your LANSAs fields. JSM uses the jar files to map the information in the XML file and map it into the RDMLX function. The binding is different depending on the direction of the XML, for example, whether it is being sent, or received. A SET is usually associated with an outbound binding, a GET with an inbound binding.

For the server to be able to receive the request, it will require an inbound binding of the request XML. The server will also require an outbound binding to generate the response XML. Assuming that the request and response XML are different (as they are in these exercises), you will create:

- an **inbound** binding jar file with the XML Binding Wizard using the request XML as input
- an **outbound** binding jar file with the XML Binding Wizard using the response XML.

This table summarizes the relationship between the input XML, the jar files and JSM command which are used for this exercise:

Input XML	Binding	Jar file	JSM Command
iiiPRO04_request.xml	Inbound	iiiPRO04_request.jar	GET
iiiPRO04_response.xml	Outbound	iiiPRO04_response.jar	SET

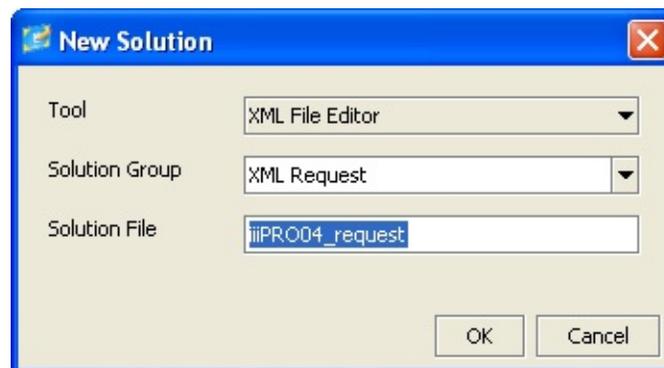
Step 1. Extend Studio Project iii Training

In this step, you will use the Integrator Studio to extend the project **iii Training** created in INT004. You will create a new solution and use the *XML Bind Wizard* to create the required folders for building your XML.

1. Locate the Integrator Studio icon by opening the LANSAs folder on your desktop.
2. Double click on the *Integrator Studio* icon. The Studio main window will appear.

You can also open Studio from the *Tools* ribbon in the LANSAs Editor.

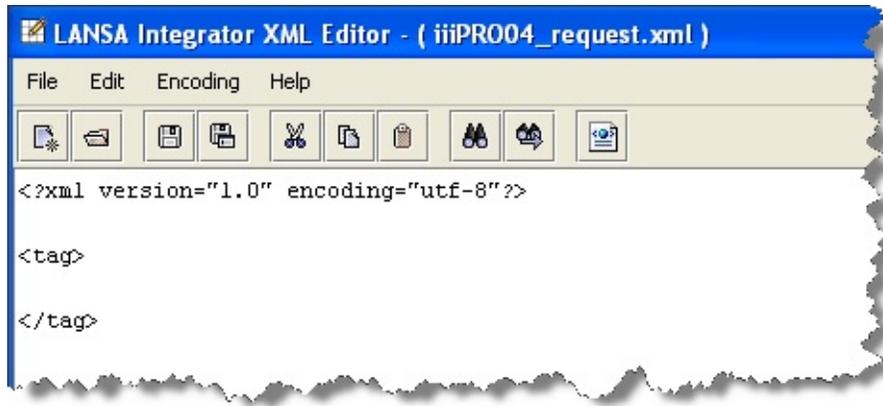
3. Select the **iii Training** project you created earlier and use the right mouse menu to *Open Project*. Your project will open in a new tab.
4. Select the solutions folder and use the right mouse menu to create *New Solution*. Select *XML File Editor* from the *Tool* dropdown. Enter a new *Solution Group* of **XML Request** and a *Solution File* name of **iiiPRO04_request**.



5. Select **Yes** in the *Confirm* dialog. A new folder called **XML Request** will be created. By default the full path will be:
c:\program files\lansa\integrator\studio\workspace\iii Training\solutions\XML Request.

Your XML file **iiiPRO04_request.xml** will be placed in this folder.

6. The *XML Editor* will open:



7. Continue to Step 2.

Step 2. Create Request XML - iiiPRO04_request.xml

In this step, since you will be writing a server function, you will write the XML to receive the request for the department description. You can use any text editor to write XML. In this case, you will use the *XML Editor* which is part of Integrator Studio.

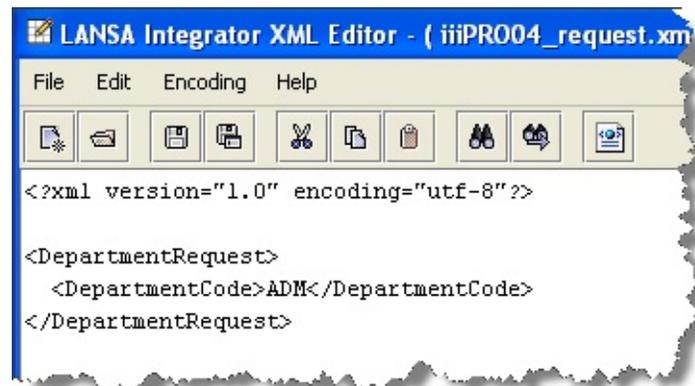
1. The open *XML editor*, contains the basic structure of an XML message. The first line in an XML file is always the XML declaration. For these exercises, you will always use the following XML declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Important Note: The `<?xml version="1.0" encoding="UTF-8"?>` statement must be the first line in the document. There should be no blank lines or characters before it, or an error will occur.

2. You must provide the department code in order to obtain the department description from the server, You will need a root tag enclosing a tag for the department code. Call the root tag **DepartmentRequest** and the department code tag **DepartmentCode**. Include a sample value of the information sent in the **DepartmentCode** tag, for example **ADM**.

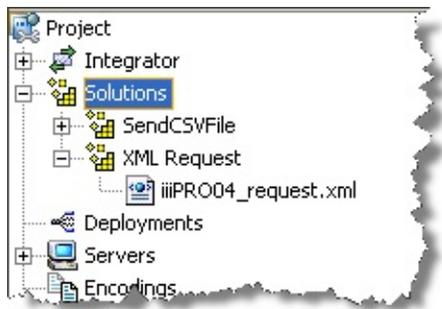
In this case, the XML is very simple and it might appear as follows:



Note that each XML tag has a start and matching end tag e.g. `<tag>` `</tag>`

3. Save the XML and select **Yes** in the *Confirm* dialog. The file was created when the editor opened. Close the *XML File Editor*.

Notice that the XML file you have just created now appears in Integrator Studio under your *XML Request* solution group.



Step 3. Create Response XML - iiiPRO04_response.xml

As noted in Step 2, you may use any text editor to create XML. In this case you will use *Notepad*. This step illustrates how you can introduce an "external" XML to your Studio project. This XML for example could have been provided by one of your company's business partners.

In this step, since you will be writing the server function, you will now write the XML to be sent as the response, containing the department description. This file will define the server's response XML. The response simply sends back the department description. The structure of the response XML is similar to the request XML. The difference between the two is the exchanged information.

Note: You could pass the Department Code back with the Department Description. In this exercise, you will pass just one field.

1. Create an *XML Response* solution group. Select *Solutions* and using the right mouse menu select *New Solution Group*. Enter **XML Response** in the dialog and click *OK*.
2. Start the *Notepad* editor in Windows. (Click on the Windows Start button, select Run, type in Notepad and click OK.)
3. Type in the XML declaration as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
```

4. Call the root tag `DepartmentResponse`. The department description tag can be called `DepartmentDescription`. Remember to include a sample value of the information sent in the `DepartmentDescription` tag, for example Administration Dept. The XML should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<DepartmentResponse>
  <DepartmentDescription>Administration Dept</DepartmentDescription>
</DepartmentResponse>
```

5. Save the XML response file in the new solution group **XML Response**. Using the File menu, select the Save As option and specify the following:

Save in locate the **XML Tutorials** directory, by default c:\program files\lansa\integrator\studio\workspace\iii

Training\solutions\XML Response

File **iiiPRO04_response.xml**

Name

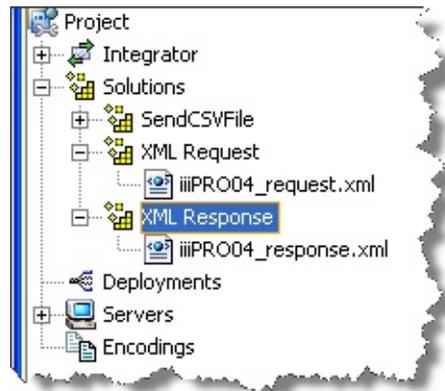
Save All Files

as

Type

6. Minimize *Notepad*.

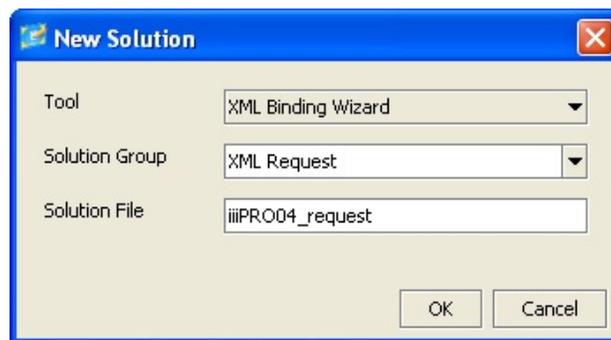
7. Return to your project in Studio and use the *View / Refresh* menu option (or F5). Your response XML is now shown in your *XML Response* folder.



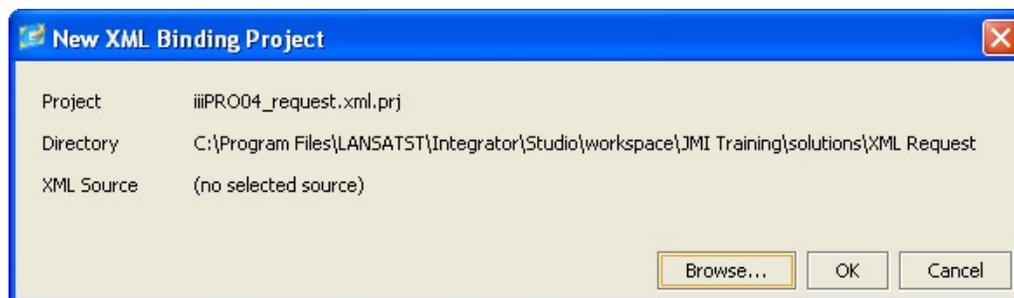
Step 4. Create iiiFN06 Server Inbound (request) binding

In this step, you will use the *XML Binding Wizard* to create the inbound request binding iiiPRO04_request.jar on the server. This jar file will bind the inbound XML for the server function to receive. You will define the mappings from the XML to your local LANSAT fields.

1. From the *Project / Solutions* node select the *New Solution* menu item.
2. A *New Solution* dialog will appear.
 - a. Select the *XML Binding Wizard* in the *Tool* dropdown.
 - b. Type or select **XML Request** in the *Solution Group* dropdown.
 - c. Type iiiPRO04_request in the *Solution File* input field. Click *OK*.

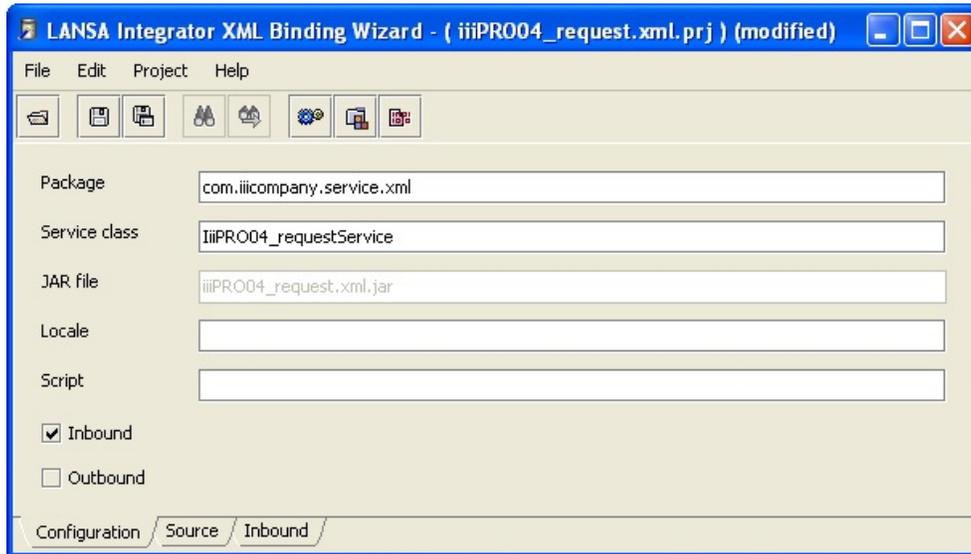


3. A new *XML Project* panel appears. To select the source XML file, click on the *Browse* button.

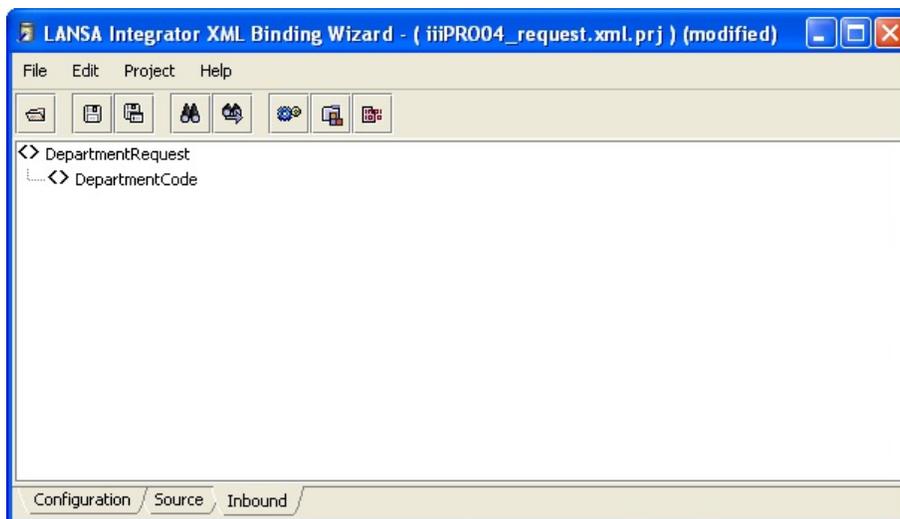


4. A *File Open* dialog will be displayed.
 - a. Click on the **iiiPRO04_request.xml** file to select it.
 - b. Click on the *Open* button.
 - c. In the *New XML Binding Project* dialog, click on the *OK* button.
5. The *XML Binding Wizard* main window will now appear.

- a. Select the Configuration tab. This defines where the java classes necessary for binding will be packaged up into jar files. Change the package name to **com.iiicompany.service.xml**
- b. You will be creating a Server to handle the inbound XML request, so click on the inbound checkbox. Notice a new Inbound tab appears.

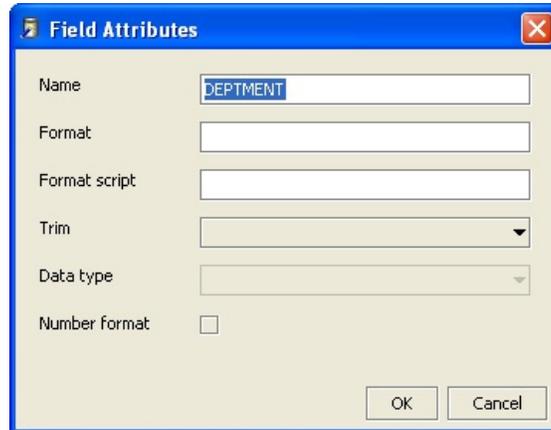


- c. Select the Source tab. You will see the XML source file added to the project.
- d. Select the Inbound tab. This is where you will map your fields.



6. Map the XML tag DepartmentCode to the LANSAs field DEPTMENT as follows:
 - a. Select the Inbound tab.

- b. Right click on the DepartmentRequest root tag and select Fragment option from the pop-up menu.
- c. Right click on Department Code tag and select Edit from the pop-up menu (or double click on it).
- d. In the *Field Attributes* dialog, enter **DEPARTMENT** for the Name and click OK



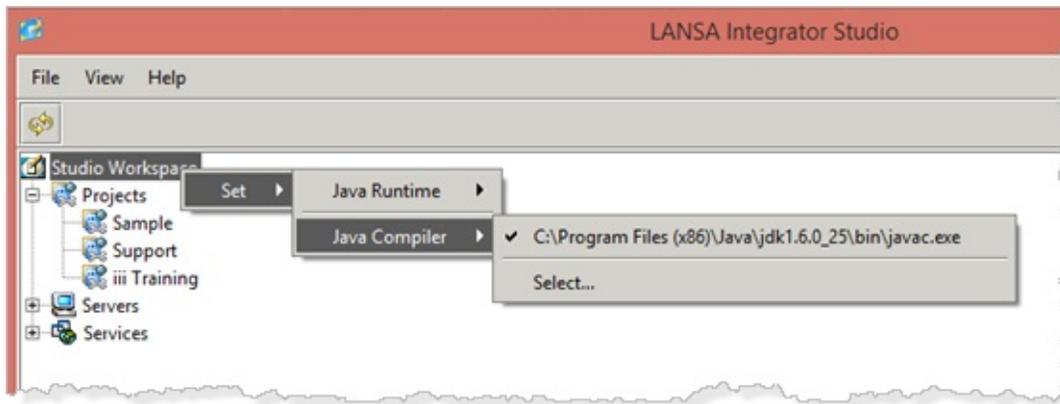
- e. Your inbound mapping should look like the following:



7. Create the inbound request binding as follows:

Note: Before you use the *Build* function for the first time, you should do the following, in order to register where Integrator Studio should locate javac.exe:

On the *Studio Workspace* tab, right click on the *Studio Workspace* item, and select *Set / Java Compiler* and then select the location for javac.exe which is shown:



b. Now continue by selecting the Project menu and choose the Build option or use the  button.



- This generates and compiles the project in order to create the necessary jar file to do the inbound request binding.
- It also generates sample text files that show you how to register this service so that it can be referenced in your RDMLX.
- Text files that contain sample RDML and RDMLX commands using your service, are also generated.

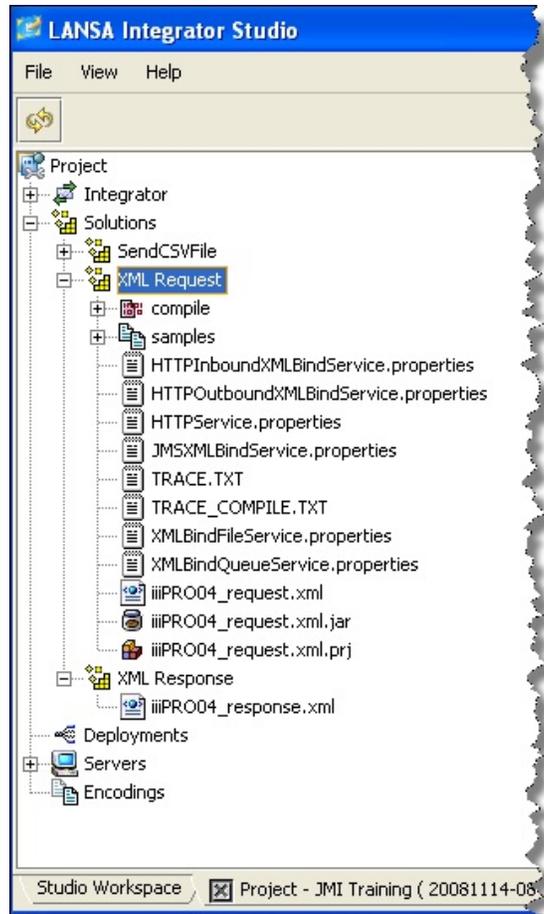
8. Click *OK* on the dialog which confirms the service has been generated and compiled.



9. Using the File menu, select the Save option. Click on Yes to save the project.

10.Exit the *XML Binding Wizard*.

11.Switch to your project in Integrator Studio. Observe that a number of folders and files have been added to your *XML Request* solution:



Step 5. Create iiiFN06 Server Outbound (response) binding

In this step, you will use the *XML Bind Wizard* to create the outbound response binding iiiPRO04_response.jar.. This jar file will bind the outbound XML from the Server to send to the client. You will define the mappings from your LANSAs fields into the XML response document.

1. With your **iii Training** project open, right click on *Solutions* to create a *New Solution*.
2. Select the *XML Binding Wizard* tool from the *Tool* dropdown.
3. Create the new *Solution* in the **XML Response** group.
4. Type iiiPRO04_response in the *Solution File* input field. Click *OK*. The *New XML Binding Project* dialog will appear.
5. Select the *Browse* button and select the source XML file.
 - a. A *File Open* dialog will be displayed.
 - b. Click on the iiiPRO04_response.xml file to select it.
 - c. Click on the *Open* button.
 - d. Click on the *OK* button.
6. The *XML Binding Wizard* main window will appear. Change the *Package name* to **com.iiicompany.service.xml**
7. You will now map the XML tag Description to the LANSAs field DEPTDESC as follows:
 - a. Select the *Outbound* checkbox (You will be creating the outbound response binding). A new *Outbound* tab will appear.
 - b. Select the *Outbound* tab. Right click on the DepartmentResponse root tag and select *Fragment* option from the pop-up menu.
 - c. Right click on *Department Description* and select *Edit* from the pop-up menu.
 - d. In the *Field Attributes* dialog, type in **DEPTDESC for the Name** and click *OK*. Your mapping should look like the following:

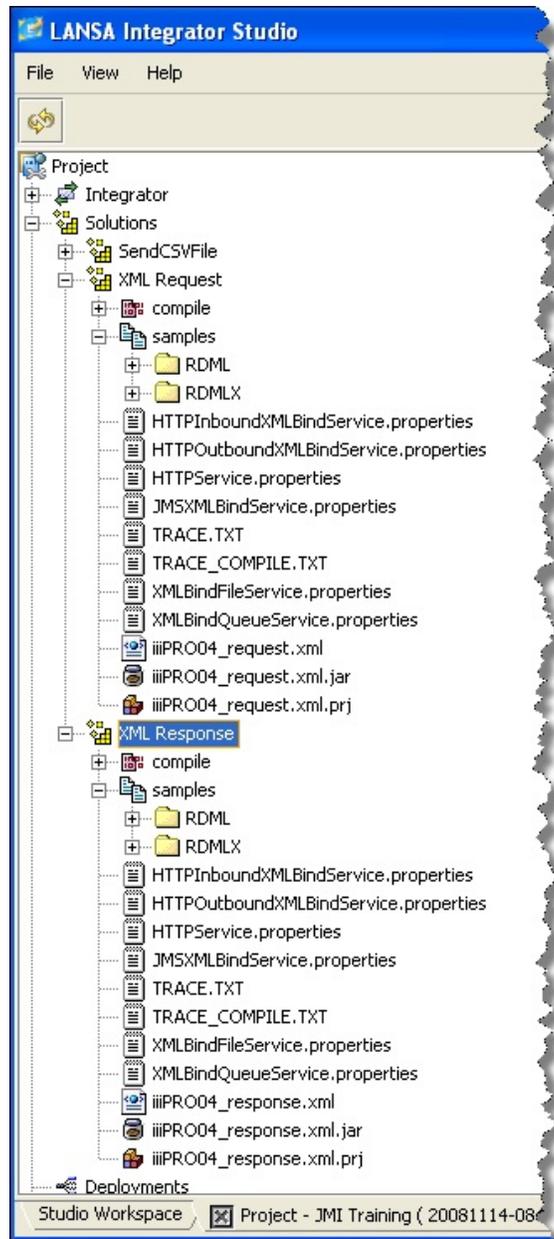


8. Create the outbound request binding as follows:
 - a. Select the Project menu and choose the Build option. This generates and compiles the project in order to create the necessary jar file to do the outbound request binding. You could also have used the *Build*  toolbar button.
9. Using the File menu, select the Save option. Alternatively, click the *Save*  toolbar button.
10. Close the *XML Wizard*.
11. Once again, in your *XML Response* solution group, note that a number of files have been generated, some of which you will use to build the rest of the application.

Step 6. Binding Deployment and Configuration

In this step, you will deploy the jar files and make other configuration changes using Integrator Studio.

1. Using the Integrator Studio, review the files in your project.
 - a. From the root Project (**iii Training**), expand the *Solutions* folder. You should see *XML Request* and *XML Response* folders.
 - b. Open the *XML Request* and *XML Response* folders. You should see your XML files and project files as well as the generated jar files for your bindings



2. In this step you will add a section for your project into the *HTTPInboundXMLBindService* properties file on the server.

In exercise INT004 you have already defined your JSM Server. This may point to **http://localhost:4561** if you are using a local web server and JSM server. If you are using an IBM i server this will point to a server name or an IP address such as **http://10.44.10.236:4564**.

- a. Select the server you are using and use the right **mouse** menu to *Open Server*.
- b. Once the JSM server has opened, you will see a second panel appear on

the right hand side. This shows all the directories and files for the JSM server instance that was just opened.

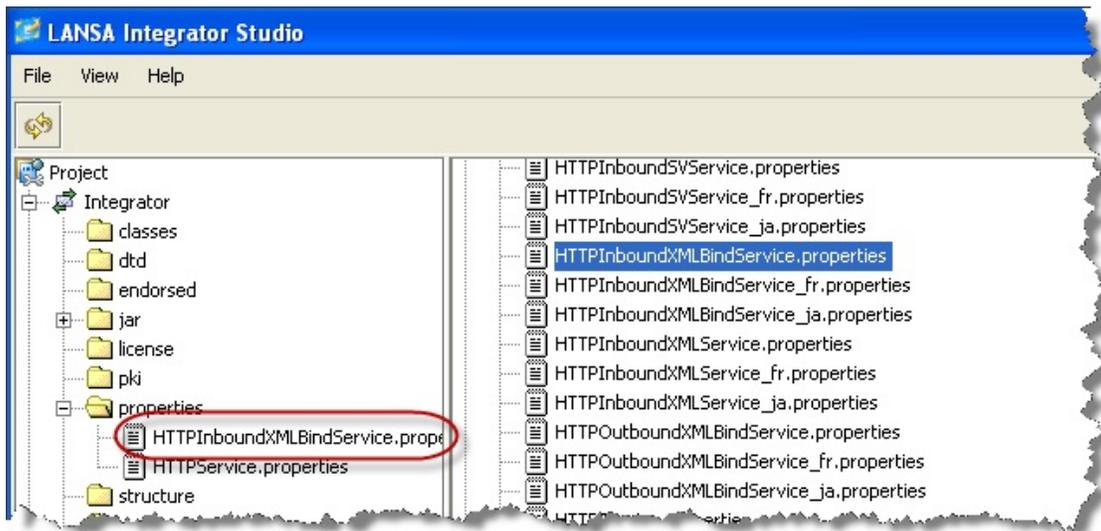
3. Your application will use the *HTTPInboundXMLBindService*.

This means that configuration entries need to be added to the properties file for this service. (The property file is named **HTTPInboundXMLBindService.properties**. Typically it is located in your JSM server in the **/instance/properties** directory).

First, you will use *Retrieve Section* to create an *HTTPInboundXMLBindService* section in your project. You will then edit this entry and the use *Publish Section* to update the properties file on the server.

- a. Open the *properties* folder in the right panel (i.e. in the JSM server instance).
- b. Scroll down and locate the *HTTPInboundXMLBindService.properties* file. Right click on the file and select the *Retrieve Section* option in the pop-up menu. A message will be displayed, "No project component properties found in properties HTTPInboundXMLBindService.properties" because there is no project specific data in the file at this time. Click *OK*.

An *HTTPInboundXMLBindService.properties* file will be created in your local *properties* folder.



- c. Open the Integrator/*properties* folder in the left panel (i.e. your local directory). Right click on the *HTTPInboundXMLBindService.properties* file and select the *Open With / Properties Editor* option from the pop-up menu.

d. An edit window will appear. The `HTTPInboundXMLBindService.properties` file will be empty. You will now add the following entries to the file:

```
service.iiipro04_request=com.IIIcompany.service.xml.IiiPRO04_requestService.archive.iiipro04_request=bindings/IIIPRO04_request.xml.jar
service.iiipro04_response=com.IIIcompany.service.xml.IiiPRO04_responseService.archive.iiipro04_response=bindings/IIIPRO04_response.xml.jar
```

The build step for the inbound and outbound HTTP XML bind service has created the entries you need. Copy these entries as follows:

- Leave the first *Property Editor (1)* open
- Expand the *XML Request* folder and open `HTTPInboundXMLBindService.properties` in the *Property editor*. Copy the code and paste it into *Property Editor (1)*. Close the second editor
- Expand the *XML Response* folder and open `HTTPOutboundXMLBindService.properties` in the editor. Copy and paste the code into *Property Editor (1)*. Close the second editor.
- Save your changes in *Property Editor (1)* and select **Yes** in the confirm dialog. Close the *Properties editor*.

Note: the java path is case sensitive. Note also that in the example shown, the build has capitalized the name of the service e.g.

IiiPRO04_responseService. This is a Java standard and cannot be changed. Throughout this workshop, you should be aware that your service names will begin with a capital letter. If this point is overlooked, your function will fail on the BIND.

e. Update the `HTTPInboundXMLBindService.properties` file on the JSM Server. Right click on your local `HTTPInboundXMLBindService.properties` file (on the left side, in *Integrator/Properties*) and select the *Publish Section* option from the pop-up menu. A message will ask you to confirm publish. Select **Yes**. You have added a section for your project to the `HTTPInboundXMLBindService.properties` file on the server.

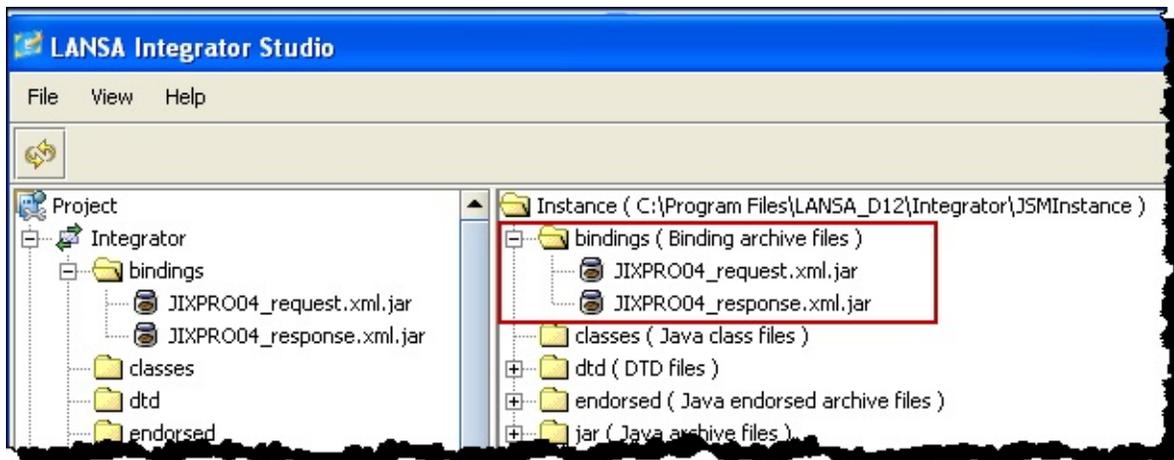
If you open the `HTTPInboundXMLBindingService.properties` file from the server and scroll to the end, it should now look like the following:

```
#!</studio-project>
#
#!<studio-project id="00000000-000000" name="unassigned">
#
# Move these entries to a new or existing studio-project section.
# Delete this unassigned section.
#
#
#!</studio-project>
#
#
#!<studio-project id="20100603-045838" name="JMI Training">
service.jixpro04_request=com.JIXcompany.service.xml.JIXPR004_requestService
service.archive.jixpro04_request=bindings/JIXPR004_request.xml.jar
service.jixpro04_response=com.JIXcompany.service.xml.JIXPR004_responseService
service.archive.jixpro04_response=bindings/JIXPR004_response.xml.jar
#!</studio-project>
#
```

4. In this step you will deploy the jar files to the server.
 - a. In the left hand panel (your project) expand the *XML Request* folder and select the **iiiPRO04_request.xml.jar** file.
 - b. Using the right mouse menu, select *Send To / Integrator Folder*.
 - c. Repeat step a and b to send the **iiiPRO04_response.xml.jar** file from *XML Response* folder to the *Integrator* folder
 - d. In the left panel, expand the *Integrator / bindings* folder.



- e. You should now see the jar files that you sent to the *Integrator/bindings* folder. One at a time, right click on each jar file in the *bindings* folder and select *Publish File*. Select **Yes** in the *Confirm* dialog.
- f. In the right window pane, expand the *\Instance\bindings* folder. You should now see the jar files that you have published.



Summary

Important Observations

- You should create your request and response in a separate solution group in order to separate the generated RDML and RDMLX code – see your Project Solution/Samples folders.
- The XMLBinding services are able to load the binding jar file from the \bindings folder at run time, and it is not necessary to restart the JSM after publishing the XMLBindings.

Tips & Techniques

- Use a naming standard that makes it easy to relate the LANSA and non LANSA components in your application.
- Start your application by creating the XML documents.
- Always remember to build your jar files for bindings and publish them to the server.
- Studio provides a convenient method of organizing your files for a specific project.
- Using a project in Studio will help you to locate changes made to the JSM properties files for a specific project. A comment tag is inserted that allows studio to locate the data specific to your project when Studio performs a retrieve or publish action.
- If you wish to transfer a complete copy of the *HTTPInboundXMLBindService.properties* file to your PC, you can use the *Download* option and then save the file to a specific directory.

What I Should Know

- How to use the Integrator Studio.
- How to use the XML Binding Wizard.
- How to build jar files for bindings.
- How to update property files on the JSM Server.
- How to publish jar files to the JSM Server.

INT006 - Department Inquiry Functions

Objectives:

- To create a server XML application that receives a Department Code and then responds by returning the Department Description.
- To create a server side RDMLX function that uses JSMDirect to receive the XML sent by the client side function using the jar files from exercise INT005.
- To use Integrator Studio to update the JSM Server database files.

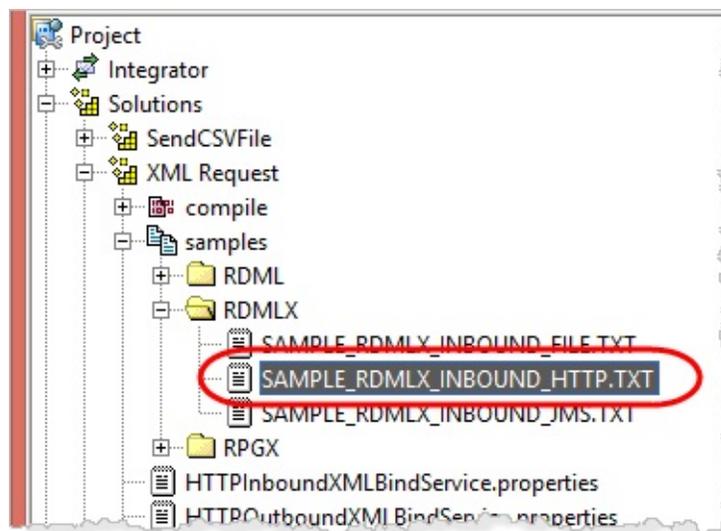
To achieve these objectives, you will complete the following:

- [Step 1. Code iiiFN06 Server GET functionality](#)
- [Step 2. Code iiiFN06 Server SET functionality](#)
- [Step 3. iiiFN06 Update JSMDirect Configuration Table](#)
- [Step 4. Create User Agent Host Configuration file](#)
- [Step 5. Test iiiFN06 function with User Agent](#)
- [Step 6. Tracing \(Optional\)](#)
- [Summary](#)

Step 1. Code iiiFN06 Server GET functionality

In this step, you will begin to write the RDMLX for the Server side GET. The server function will receive the XML from the client.

1. Create a new LANSAPROCESS named iiiPRO04 XML Tutorial, where iii is your unique 3 characters. (If the process already exists, select a different set of characters for iii.)
2. Create a new function named **iiiFN06 Department Server**, belonging to process **iiiPRO04** (where **iii** is your unique 3 characters). Create the function without using a template and make sure that Enabled for RDMLX is checked.
3. Switch to your project in Integrator Studio. In the *XML Request* solution expand the *samples/RDMLX* folder and open **SAMPLES_RDMLX_INBOUND_HTTP.TXT** in the Studio text editor. Simply double clicking on it will open the text editor.



4. Copy and paste all the code into your function iiiFN06. Replace existing code.

Note: The fields shown as comments at the top of this code should already exist in the Repository, if not, create them.

5. Locate the GROUP_BY command towards the top of the code and change its name to DEPTREQ. This GROUP_BY is simply reference so that a value can be mapped into field DEPARTMENT.
6. At this stage, the server function now has the information from the XML document. Save your changes.

Step 2. Code iiiFN06 Server SET functionality

In this step, you will write the RDMLX to access the database file DEPTAB to FETCH the department description. You will write the RDMLX for the server side SEND to respond to the client request.

1. Working with function iiiFN06, after the **GET** from the DEPARTMENTREQUEST fragment, remove the block highlighted below:

```
* Get fragment - DEPARTMENTREQUEST
Change Field(#JSMXCMD) To('GET FRAGMENT(DEPARTMENTREQUEST)
SERVICE_EXCHANGE(*FIELD)')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)

* Bind service to create HTTP response content
Change Field(#JSMXCMD) To('BIND SERVICE( <<<outbound.class>>> ) TYPE(*OUTBOUND)')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)
```

then write the RDMLX code to clear the DEPTDESC field, FETCH the field DEPTDESC (department description) from file DEPTAB with key DEPTMENT (department code).

Check the I/O status of the FETCH operation. If it is not *OKAY, change the department description to the literal 'Department not found'.

The RDMLX code might appear as follows:

```
#DEPTDESC := *NULL
FETCH FIELDS(#DEPTDESC) FROM_FILE(DEPTAB) WITH_KEY(#DEPTMENT)
IF_STATUS IS_NOT(*OKAY)
#DEPTDESC := 'Department Not Found'
ENDIF
```

2. You now need to add code to create the outbound HTTP content. You can take this code fragment from the RDMLX generated by Studio for your *XML Response* solution. Expand the folder *XML Response/samples/RDMLX* and open *SAMPLE_RDMLX_OUTBOUND_HTTP.txt* in the text editor. Copy the highlighted code into function iiiFN06 immediately following the comment:

```
* <<< Outbound binding logic goes here >>>.
```

```

LANSA Integrator Text Editor - ( SAMPLE_RDMLX_OUTBOUND_HTTP.TXT )
File Edit Encoding Help
* Load service
CHANGE FIELD(#JSMXCMD) TO('SERVICE_LOAD SERVICE(HTTPOutboundXMLBindService) TRACE(*YES)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* Bind service to create HTTP request content
CHANGE FIELD(#JSMXCMD) TO('BIND SERVICE(IIIPRO04_RESPONSE) TYPE(*OUTBOUND)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* Set fragment - DEPARTMENTRESPONSE
CHANGE FIELD(#JSMXCMD) TO('SET FRAGMENT(DEPARTMENTRESPONSE) SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* Write content
CHANGE FIELD(#JSMXCMD) TO('WRITE INDENT(*YES) BINDTRACE(*YES)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

```

3. In the CHECK subroutine, add an ABORT command to the IF..ENDIF condition so that the program ends if an error has occurred.
4. Compile the function.
5. If you are using an IBM i JSM server, check the function into the IBM i and compile it.

Your finished RDMLX code might appear as follows:

```

FUNCTION OPTIONS(*DIRECT)
* The following fields are used by the xml binding map
* #DEPTMENT
* The following fragments are used by the xml binding map
GROUP_BY NAME(#DEPTREQ) FIELDS(#DEPTMENT)
* Open service
USE BUILTIN(JSMX_OPEN) TO_GET(#JSMXSTS #JSMXMSG
#JSMXHDLE1)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
* Load service
CHANGE FIELD(#JSMXCMD) TO('SERVICE_LOAD
SERVICE(HTTPInboundXMLBindService) SERVICE_CONTENT(*HTTP)
TRACE(*YES)')

```

```

USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
* Bind service to read HTTP request content
CHANGE FIELD(#JSMXCMD) TO('BIND
SERVICE(IIIPRO04_REQUEST) TYPE(*INBOUND)
BINDTRACE(*YES)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
* Get fragment - DEPTREQ
CHANGE FIELD(#JSMXCMD) TO('GET
FRAGMENT(DepartmentRequest) SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
* handle request for department description
#deptdesc := *blanks
FETCH FIELDS(#deptdesc) FROM_FILE(deptab) WITH_KEY(#deptment)
IO_ERROR(*NEXT) VAL_ERROR(*NEXT)
IF_STATUS IS_NOT(*OKAY)
#deptdesc := ('Department not found')
ENDIF
* <<< Outbound binding logic goes here >>>
* Bind service to create HTTP request content
CHANGE FIELD(#JSMXCMD) TO('BIND
SERVICE(IIIPRO04_RESPONSE) TYPE(*OUTBOUND)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
* Set fragment - DEPARTMENTRESPONSE
CHANGE FIELD(#JSMXCMD) TO('SET
FRAGMENT(DEPARTMENTRESPONSE)
SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1

```

```

#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)
* Write content
CHANGE FIELD(#JSMXCMD) TO('WRITE INDENT(*YES)
BINDTRACE(*YES)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)
* Send HTTP response content
CHANGE FIELD(#JSMXCMD) TO('SEND')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)
* Close service
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1)
TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)
* Check routine
SUBROUTINE NAME(CHECK) PARM(#JSMXSTS *RECEIVED)
(#JSMXMSG *RECEIVED))
IF COND('#JSMXSTS *NE OK')
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1)
TO_GET(#JSMXSTS #JSMXMSG)
ABORT
ENDIF
ENDROUTINE

```

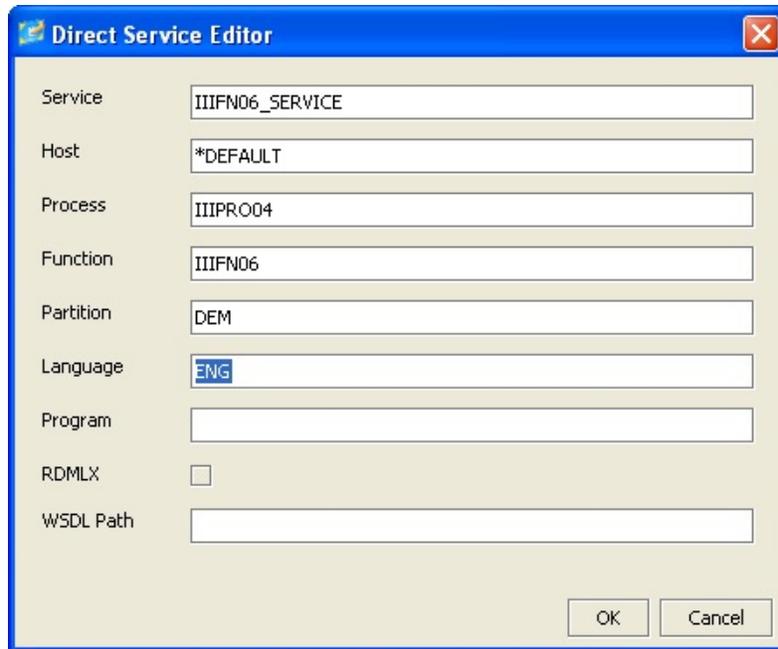
Step 3. iiiFN06 Update JSMDirect Configuration Table

In this step, you will make changes to the *Configuration Table*. When a service request arrives at the host, JSM checks the file DC_W29.txt on Windows or database file DC@W29 on the IBM i to identify the function to execute. The key used by JSM is the **Service Name** and the **Host: Port**. The *Service Name* is the part of the URL keyword value that comes after the question mark entered as the value for the URI parameter: **iiiFN06_SERVICE**.

1. Switch to Integrator Studio and select the *Studio Workspace* tab. Expand the *Services* folder and select the service created in INT004. If you are using an IBM i JSM Server, this may look similar to the following:



2. Right click on this service and select *Open Service*. A list of the existing services will be displayed. This list will depend on what services have already been defined for this server.
2. Right click anywhere on a white space and select *New* to create a new service entry.
3. A *Direct Service Editor* window will appear.



The image shows a dialog box titled "Direct Service Editor" with a blue title bar and a close button (X) in the top right corner. The dialog contains several input fields and a checkbox:

Service	IIIFN06_SERVICE
Host	*DEFAULT
Process	IIIPRO04
Function	IIIFN06
Partition	DEM
Language	ENG
Program	
RDMLX	<input type="checkbox"/>
WSDL Path	

At the bottom right of the dialog, there are two buttons: "OK" and "Cancel".

4. Add a new entry with the values shown, substituting your value for **iii**. Note that the partition may be different for your course.
5. Click *OK* to save your changes and close the *Direct Services* tab. The file DC@W29 for the correct JSM Server instance has been updated on the IBM i.
 - i. If you are using http://localhost, the file DC_W29.txt has been updated.

Step 4. Create User Agent Host Configuration file

In this step, you will set up the *User Agent* host configuration file to test your server function. The User Agent will act as a client querying the server you have created.

1. You need to copy the `SendCSVFile.lih` file from the `SendCSVFile` solution group to your XML Request solution group.

With your project open in Studio, expand the folder *Solutions / SendCVSFile*. Select and right click the file `SendCVSFile.lih`. Select *Copy File*.



2. Expand your *XML Request* solution, Select *XML Request*, right click and select *Paste File*. Click *OK* to confirm. The `SendCVSFile.lih` file will be copied to your *XML Request* folder. Select the file and use the right mouse menu to *Rename File*. Change the file name from **iiiPRO04_client** to **iiiPRO04_client_test.lih**.

3. You need to modify the User Agent file to connect to the server and service you have created. Right-click on `iiiPRO04_client_test.lih` and select *Open With / Properties Editor*. Make the following replacements:

Old Line

name=iiiFN05_SERVICE

xml.uri=/cgi-bin/jsmdirect?

New Line

name=iiiFN06_SERVICE

xml.uri=/cgi-bin/jsmdirect?

orderxml

iiiFN06_SERVICE

If you are using a local web server and JSM server the uri line will be:
xml.uri=/cgi-bin/jsmdirect.exe?iiiFN06_SERVICE

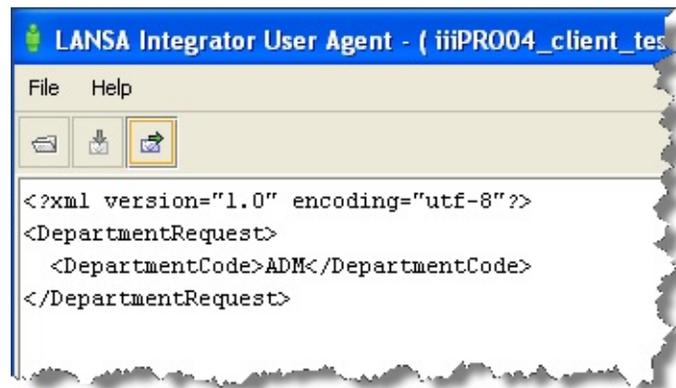
4. You have completed the creation of your User Agent host configuration file.
Save and close iiiPRO04_client_test.lih.

Step 5. Test iiiFN06 function with User Agent

In this step, you will test your Department Server by running the User Agent file you created in Step 4.

1. You will load your iiiPRO04_client_test.lih into the User Agent and send iiiPRO04_request.xml to the **iiiFN06_SERVICE** service on the JSM server.
 - a. Double click iiiPRO04_client_test.lih to open the User Agent using this host configuration file. Your host configuration file properties will be used to determine where to send the XML source.
 - b. Now, select the xml file to send to the server. You will send the *Request* document created earlier in INT005. Use the  *Open Source* button.
 - c. The *Open* dialog will default to the *XML Request* folder. Select iiiPRO04_request.xml and select the *Open* button.

The source is now displayed by the User Agent. Review the XML in the *Source* tab:

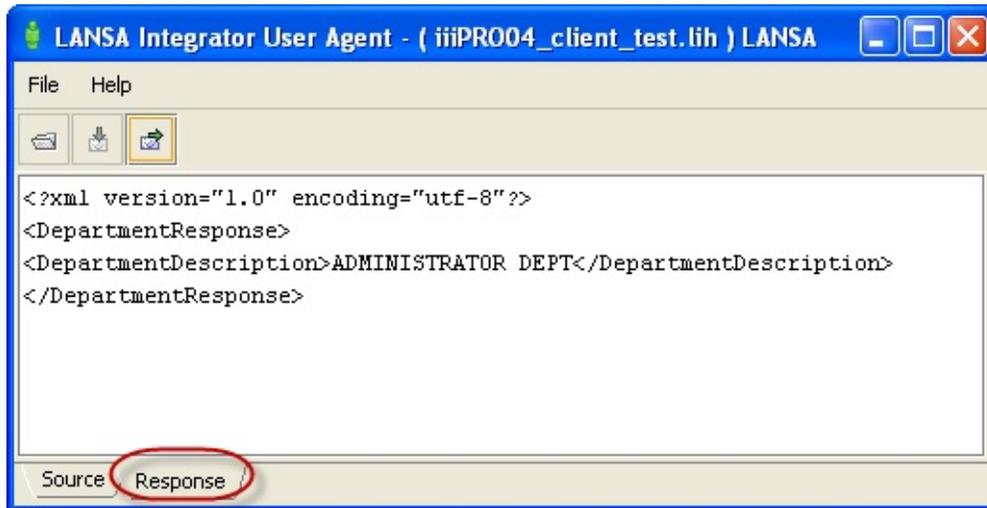


- d. You are now ready to send your request to the server. Select the  *Send* button to send the xml file.
 - e. Press *Yes* to confirm that you wish to send the specified File to the specified Host using that *Transport* method. Press *OK* to confirm the completed transaction.



The XML file has been sent to the server and will be processed by function iiiFN06.

The *Response* tab should look like the following:



The iiiFN06 function, called via the **iiiFN06_SERVICE** service, has processed the iiiPRO04_request.xml and returned an appropriately populated version of iiiPRO04_response.xml.

2. Try other valid Department codes, such as FLT (FLEET ADMINISTRATION) or MKT (MARKETING DEPARTMENT). To do this, select and right-click iiiPRO04_request.xml, select *Open With, XML Editor*, and change the ADM text to the desired test data. The Department Description in the Response will change accordingly.

Invalid Department codes should return a Department Description of **DEPARTMENT NOT FOUND**.

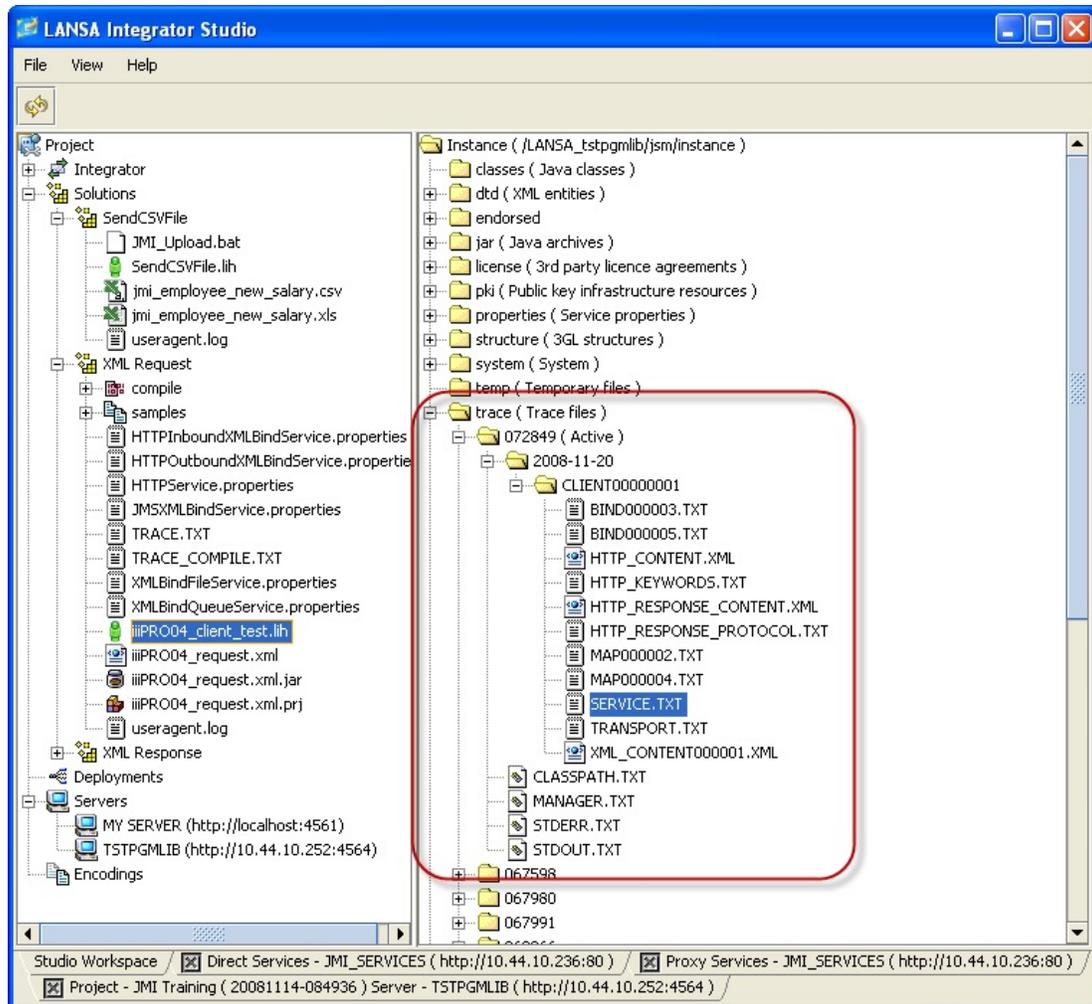
Step 6. Tracing (Optional)

In this step, you will use the Integrator Studio to review the different trace files that are created while executing your functions. This is an optional step. In order to use tracing, tracing must have been turned on. The tracing setting can be configured in the `manager.properties` file or as a parameter of the `SERVICE_LOAD` Command.

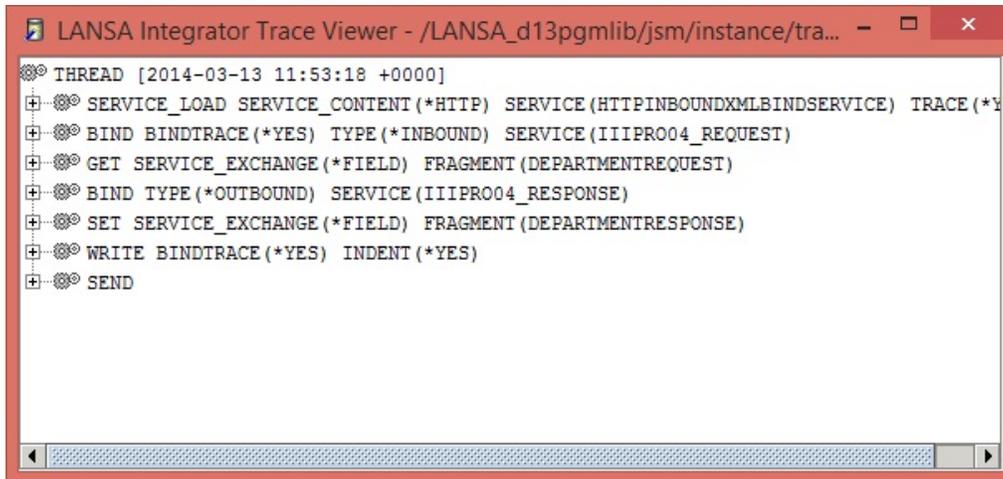
1. Review function `iiiFN06` in the Visual LANSA editor. The code generated by Studio, includes `TRACE(*YES)` on the `SERVICE_LOAD` command. Note that trace should be turned off in your production application, to avoid overheads.

```
* Load service
CHANGE FIELD(#JSMXCMD)
  TO('SERVICE_LOAD SERVICE(HTTPInboundXMLBindService) SERVICE_CONTENT(*HTTP) TRACE(*YES)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARS(#JSMXSTS #JSMXMSG)
```

2. Run your application
3. On the *Project* tab in Studio, open the server. Expand the *trace (Trace Files)* folder on the JSM Server panel.



4. Look for the *active* job number and open this folder.
5. Open the client folders and review the contents of the different trace files.
Note: The Service and Transport trace files can be opened with a *Trace Viewer*.



```
LANSA Integrator Trace Viewer - /LANSA_d13pgmlib/jsm/instance/tra...
THREAD [2014-03-13 11:53:18 +0000]
SERVICE_LOAD SERVICE_CONTENT (*HTTP) SERVICE (HTTPINBOUNDXMLBINDSERVICE) TRACE (*Y
BIND BINDTRACE (*YES) TYPE (*INBOUND) SERVICE (IIIPRO04_REQUEST)
GET SERVICE_EXCHANGE (*FIELD) FRAGMENT (DEPARTMENTREQUEST)
BIND TYPE (*OUTBOUND) SERVICE (IIIPRO04_RESPONSE)
SET SERVICE_EXCHANGE (*FIELD) FRAGMENT (DEPARTMENTRESPONSE)
WRITE BINDTRACE (*YES) INDENT (*YES)
SEND
```

6. To download the trace files, right click one of the client folders and select the *Download* option from the pop-up menu. All trace files will be placed into a zip file and a new *Download* tab will be shown in Integrator Studio. Select a file on the *Download* tab to save it. The downloaded file could now be emailed to another developer when troubleshooting a problem.

Summary

Important Observations

- Server side functions start as different jobs so they require JSM to be opened and closed.
- Server side functions do not unload the service.

Tips & Techniques

- Use a naming standard that makes it easy to relate the LANSA and non LANSA components in your application.
- Begin your application development by creating the XML forms.

What I Should Know

- How to code server-side JSM functions.
- How to use and configure JSMDirect.
- How to modify the HTTPInboundXMLBindService.properties file.
- How to use Integrator Studio to update the JSMDirect Configuration Table.
- How to use the User Agent to test your server-side functions.

INT007 - Department List Inquiry

Objectives:

- To create a server XML application that receives a list of Department Codes and a simple field which then responds back with a list of Department Descriptions and a simple field.
- To create a server side RDMLX function that uses JSMDirect to receive the XML sent by the client side function.

To achieve this objective, you will complete the following:

- [Concepts](#)
- [Step 1. Create XML iiiPRO05_request.xml](#)
- [Step 2. Create XML iiiPRO05_response.xml](#)
- [Step 3. Create iiiFN07 Server inbound \(request\) binding](#)
- [Step 4. Create iiiFN07 Server outbound \(response\) binding](#)
- [Step 5. Binding Deployment and Configuration](#)
- [Step 6. Code iiiFN07 Server GET functionality](#)
- [Step 7. Code iiiFN07 Server Database Logic and SET functionality](#)
- [Step 8. Update Configuration Table for iiiFN07](#)
- [Step 9. Create User Agent file for iiiFN07](#)
- [Step 10. Test iiiFN07 function with User Agent](#)
- [Summary](#)

Concepts

In this exercise, the client and server functions will exchange:

- a list of Department codes
- a simple field (in order to show a combined XML structure).

These requirements mean that the XML file requires a slightly different structure compared with the XML used in INT006.

The following table summarizes the relationship between the input XML, binding JARs, and JSM commands which are used for this exercise:

Input XML	Binding	JAR file	JSM Command
iiiPRO05_request.xml	Inbound	iiiPRO05_request.jar	GET
iiiPRO05_response.xml	Outbound	iiiPRO05_response.jar	SET

Step 1. Create XML iiiPRO05_request.xml

In this step, you will write the XML to get the request for department descriptions. You can use any text editor to write XML. You will start by editing the XML used in INT005.

1. Begin in Integrator Studio with your **iii Training** project open. Right click on solutions and create a *New Solution Group XML List Request*.
2. Expand the **XML Request** folder which you created in INT005, and double click on iiiPRO04_request.xml to open it in the XML Editor.

Your XML should appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<DepartmentRequest>
  <DepartmentCode>ADM</DepartmentCode>
</DepartmentRequest>
```

3. Modify the name of root tag `<DepartmentRequest>` and its closing tag `</DepartmentRequest>` to refer to a list of departments.

Your XML might appear as follows:

```
<DepartmentListRequest>
  <DepartmentCode>ADM</DepartmentCode>
</DepartmentListRequest>
```

4. The simple field you are going to pass is just an alphanumeric string that the server function is going to reverse and then return back to you. Within the root tag, add a tag for the simple field. Include a sample value in the new tag.

Your XML might appear as follows:

```
<DepartmentListRequest>
  <ReverseString>ABCDEFG</ReverseString>
  <DepartmentCode>ADM</DepartmentCode>
</DepartmentListRequest>
```

5. Next, you have to change the XML so that the department code occurs multiple times because it is a list. You will make the department code tag become the parent tag of the actual code tag.
 - a. Delete the ADM literal between the opening `<DepartmentCode>` and

closing `</DepartmentCode>` tags and move the closing `</DepartmentCode>` to the next line so that it aligns with the opening `<DepartmentCode>`.

- b. Rename the `<DepartmentCode>` tag and call it just `<Department>`. (This is not necessary but it will eliminate redundancy in the terminology as you are going to add a child tag with the actual department.)

Your XML might appear as follows:

```
<DepartmentListRequest>
  <ReverseString>ABCDEFGF</ReverseString>
  <Department>
</Department>
</DepartmentListRequest>
```

6. Insert a child tag of `<Department>` with a sample value of ADM. Call the child tag `<Code>`. Remember that all tags require a closing tag. Make sure you indent the child tag to make for easier reading.

Your XML might appear as follows:

```
<DepartmentListRequest>
  <ReverseString>ABCDEFGF</ReverseString>
  <Department>
    <Code>ADM</Code>
  </Department>
</DepartmentListRequest>
```

7. Finally, you have to modify the XML to represent the multiple occurrences of the Department code. There are a several ways to accomplish this task. One method is to repeat the child tags `<Code></Code>`:

```
<Department>
  <Code>ADM</Code>
  <Code>FLT</Code>
</Department>
```

Alternatively, for this exercise, you will simply repeat the `<Department></Department>` tag making another block. This will make the request and response XML structurally the same.

Your XML might appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<DepartmentListRequest>
  <ReverseString>ABCDEFGFG</ReverseString>
  <Department>
    <Code>ADM</Code>
  </Department>
  <Department>
    <Code>FLT</Code>
  </Department>
</DepartmentListRequest>
```

Remember that the line `<?xml version=>` must be the first line and have no other leading characters or lines.

8. Using the File menu, select the Save As option as specify the file as `iiiPRO05_request.xml`. Save it to the folder **XML List Request**.

Step 2. Create XML iiiPRO05_response.xml

In this step, you will write the XML to SEND (server) and RECEIVE (client) the response with the department descriptions. The client function only requires a response consisting of a list of department descriptions; however, the department code and the department description will also be sent back in order to make the RDMLX code smaller and simpler. The structure of the response XML is the same as the request XML.

1. In Studio, select *Solutions* in your project tab and use *New Solution Group* to create a folder **XML List Response**.
2. Working with your opened iiiPRO05_request.xml, modify the root tag to define the response XML. Call the root DepartmentListResponse. Remember to modify the closing tag as well.

Your XML might appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<DepartmentListResponse>
  <ReverseString>ABCDEFGFG</ReverseString>
  <Department>
    <Code>ADM</Code>
  </Department>
  <Department>
    <Code>FLT</Code>
  </Department>
</DepartmentListResponse>
```

3. Modify your XML so that the department description is also exchanged. Insert a department description tag within the <Department> tag. Call the tag <Description>.

Your XML might appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<DepartmentListResponse>
  <ReverseString>ABCDEFGFG</ReverseString>
  <Department>
    <Code>ADM</Code>
    <Description>Administration Dept</Description>
  </Department>
```

```
<Department>
  <Code>FLT</Code>
  <Description>Fleet Administration</Description>
</Department>
</DepartmentListResponse>
```

4. Using the File menu, select the Save As option as specify the file as iiiPRO05_response.xml. Save it to the folder **XML List Response**.

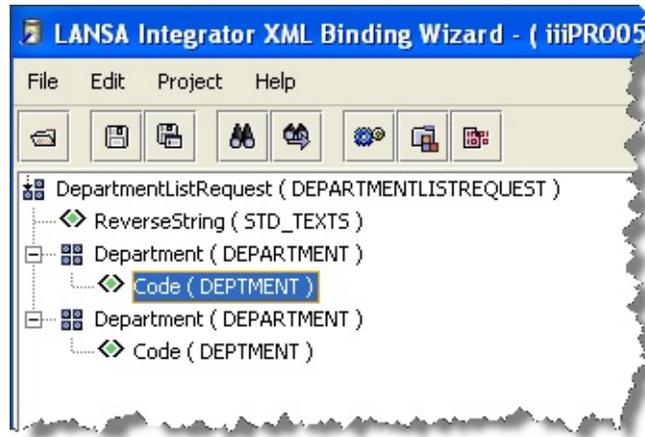
Step 3. Create iiiFN07 Server inbound (request) binding

In this step, you will use the XML Binding Wizard to create the inbound request binding `iiiPRO05_request.jar` which will bind the inbound XML that the server receives.

1. From the Studio *Project* tab, select the *XML List Request* solution. Use the right mouse menu to create a *New Solution*, a dialog will appear.
 - a. Select the *XML Binding Wizard* from the *Tool* dropdown.
 - b. *XML List Request* should already be selected in the *Group* dropdown.
 - c. Type `iiiPRO05_request` in the *Solution File* input field. Click *OK*.
2. A new *XML Binding Project* panel appears. To select the source XML file, click on the *Browse* button. A *File Open* dialog will be displayed.
 - a. Select the `iiiPRO05_request.xml` file and click *Open* to select it. Click *OK* and the *XML Binding Wizard* main window will now appear. Change the *Package name* to `com.iiicompany.service.xml`.
 - b. Check the *Inbound* checkbox. An *Inbound* tab will now appear at the bottom of the window.
3. You need to map both the list of Department Codes and the ReverseString field. First, map the XML tag ReverseString to the LANSAs field `STD_TEXTS` as follows:
 - a. Select the *Inbound* tab.
 - b. Right click on the `DepartmentListRequest` root tag. Select *Fragment* from the pop-up menu.
 - c. Right click on `ReverseString` tag and select *Edit* from the pop-up menu (or double click on it).
 - d. In the *Field Attributes* dialog, type in **STD_TEXTS** for the Name and click *OK*.
 - e. Right click on either of the `Department` tags and select *List*. This will define both tags, as well as any sub-tags in the hierarchy, as members of a list.
 - f. Expand the `Department` tags to see their sub-tags. Right click or double click on either of the `Code` tags to edit them.
 - g. In the *Field Attributes* dialog, type in **DEPARTMENT** for the Name and

click OK. Notice how editing one of the sub-tags in the list automatically edits all occurrences of that tag in the list.

4. Your XML mappings are complete. Your *XML Binding* should look like the following:



5. Now, create the inbound request jar. Select the Project menu and choose the Build option or use the  Build toolbar button.
6. Using the File menu, select the Save option or use the  Save toolbar button. Click on Yes to save the project.
7. Exit the *XML Binding Wizard*.

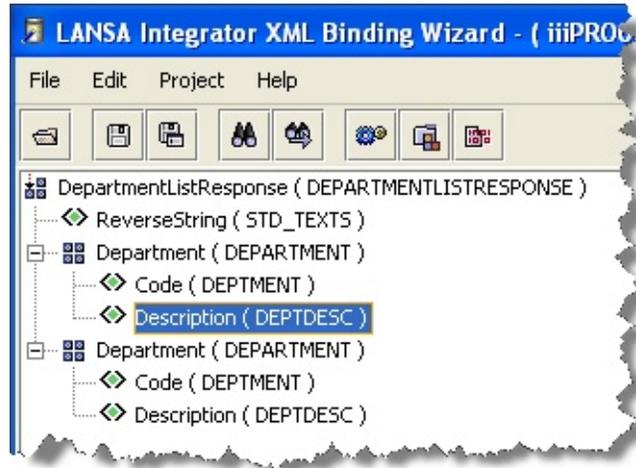
Step 4. Create iiiFN07 Server outbound (response) binding

In this step, you will use the *XML Binding Wizard* to create the outbound response binding `iiiPRO05_response.jar`. This jar file will bind the outbound XML from the Server to send to the client. You will define the mappings from your LANSAs fields into the XML response document.

1. From the *Project* tab select the *XML List Response* solution group and use the right mouse menu option *New Solution*, a dialog will appear.
 - a. Select the *XML Binding Wizard* in the *Tool* dropdown.
 - b. *XML List Response* will already be selected in the *Group* dropdown.
 - c. Type `iiiPRO05_response` in the *Solution File input* field. Click *OK*.
2. A new *XML Binding Project* dialog appears. To select the source XML file, click on the *Browse* button. A *File Open* dialog will be displayed.
3. Select the `iiiPRO05_response.xml` file and click *Open* to select it. Click *OK* and the *XML Binding Wizard* main window will now appear. Change *Package name* to `com.iiicompany.service.xml`.
4. Check the *Outbound* checkbox. An *Outbound* tab will now appear at the bottom of the window.
5. The outbound mapping is nearly identical to the inbound mapping. The only difference is the naming of the root tag, and the addition of a description node within the list.
 - a. Select the *Outbound* tab.
 - b. Right click on the `DepartmentListResponse` root tag. Select *Fragment* from the pop-up menu.
 - c. Right click on `ReverseString` tag and select *Edit* from the pop-up menu (or double click on it).
 - d. In the *Field Attributes* dialog, type in **STD_TEXTS** for the Name and click *OK*.
 - e. Right click on either of the `Department` tags and select *List*.
 - f. Expand the `Department` tags to see their sub-tags. Right click or double click on either of the `Code` tags to edit them.
 - g. In the *Field Attributes* dialog, type in **DEPARTMENT** for the Name and click *OK*.

- h. Right click or double click on either of the Description tags to edit them.
- i. In the *Field Attributes* dialog, type in **DEPTDESC** for the Name and click OK.

Your *XML outbound binding* should look like the following:



- 6. Your XML mappings are complete. Now, generate and compile the outbound response service. Select the Project menu and choose the Build option.
- 7. Using the File menu, select the Save option. Click on Yes to save the project.
- 8. Exit the *XML Binding Wizard*.

Step 5. Binding Deployment and Configuration

In this step, you will deploy the jar files and update the HTTPInboundXMLBindService properties file using Integrator Studio. In a previous exercise INT004, you have created a local properties section for this service. You will now update this and then *Publish* it to the server, as before.

1. With your *Project* tab open in Studio, Expand the *Integrator / properties* folder and double click on the HTTPInboundXMLBindService.properties file to open it in the properties editor.

The HTTPInboundXMLBindService.properties file should have the following entries from INT004:

```
service.iiipro04_request=com.iiicompany.service.xml.IiiPRO04_requestSe  
service.archive.iiipro04_request=bindings/iiipro04_request.xml.jar  
service.iiipro04_response=com.iiicompany.service.xml.IiiPRO04_responseSer  
service.archive.iiipro04_response=bindings/iiipro04_response.xml.jar
```

- a. You will now add the following lines to the file:

```
service.iiipro05_request=com.iiicompany.service.xml.IiiPRO05_requestServic  
service.archive.iiipro05_request=bindings/iiipro05_request.xml.jar  
service.iiipro05_response=com.iiicompany.service.xml.IiiPRO05_responseSer  
service.archive.iiipro05_response=bindings/iiipro05_response.xml.jar
```

Do this as follows:

- b. Leave the first properties file open in the *Properties Editor (1)*.
- c. Expand the folder *XML List Request* and open the file HTTPInboundXMLBindService.properties in the *Properties editor*. Copy and paste this code into *Properties Editor (1)*. Close the second properties editor.
- d. Repeat these steps to copy the code from *XML List Response / HTTPInboundXMLBindService.properties* into *Properties Editor (1)*.
- e. Close the second editor. Save your changes in *Properties Editor (1)* and

close it.

Note: The java path is case sensitive.

2. Now you need to update the server's version of `HTTPInboundXMLBindService.properties` with your modifications.
 - a. Your JSM Server instance is probably already open on the right hand side from a previous step. If not, open it by double clicking on the server in your *Project* tab.
 - b. In the *Integrator / Properties folder*, right click on your local `HTTPInboundXMLBindService.properties` file and select the *Publish Section* option from the context menu.
 - c. A message will ask you to confirm publish. Press *Yes*.

The `HTTPInboundXMLBindService.properties` file on the server now defines what resource to use for this exercise.

3. Deploy the jar files.

You must copy the jar files from your PC into the appropriate JSM server directory. You will send the jar files to the directory `\instance\bindings`.

- a. In the *XML List Response* solution, right click on `iiiPRO05_response.xml.jar` and select *Send To /, Integrator Folder* from the context menu. Select *Yes* when asked if you want to copy the file over. Note that alternatively, you could simply drag and drop the `iiiPRO05_response.jar` file into the *Integrator/bindings* folder.
- b. In the *XML List Request* folder, repeat step a. for `iiiPRO05_request.xml.jar`.
- c. In the left panel, expand the *Integrator / bindings* folder. One at a time, right click on each jar file and select *Publish File*. Select *Yes* when prompted.
- d. You will not need to reboot your JSM server since the JSM will retrieve the `XMLBinding` jar files from the `/bindings` folder...

Step 6. Code iiiFN07 Server GET functionality

In this step, you will begin to write the RDMLX for the Server side GET. The client will have sent a request with a list of Department Codes to the server. The server function will receive the XML form and map the data into program variables.

1. Create a new LANSAPROCESS named iiiPRO05 JSM XML List Tutorial, where iii is your unique 3 characters. (If the process already exists, select a different set of characters for iii.).
2. Create a new function named iiiFN07 Department List Server, belonging to process iiiPRO05. The function must be RDMLX enabled. Create it without using a template.
3. With your project **iii Training** open in Studio, expand the *Solutions / XML List Request / Samples / RDMLX* folder and double click on **SAMPLE_RDMLX_INBOUND_HTTP.TXT** to open it in the text editor. Copy the code into your function iiiFN07, replacing existing code.
4. Edit the RDMLX code of function iiiFN07.
 - a. Change the GROUP_BY name DEPARTMENTLISTREQUEST to LISTREQ. This Group_By is simply a reference for mapping the field STD_TEXTS.
 - b. Use the *Replace* function to change all occurrences of #DEPARTMENT to #DEPTMEN_W. Use **Match whole words only**. This is a working list for the received list of department codes. The code generator is currently not respecting RDML name length limits. This should change 2 occurrences.
5. Save this function. It is not yet complete. It will be finished in the next step.

At this stage, the server function now has the information from the XML document.

Step 7. Code iiiFN07 Server Database Logic and SET functionality

In this step, you will write the RDMLX to access the database file DEPTAB to FETCH the department description, and write the RDMLX for the server side SET.

1. Following the working list definition DEPTMEN_W, define a second working list DEPTDES_W containing field DEPTESC. This is the list which will be returned. Add code to clear both lists. Your code should look like the following:

* The following lists are used by the xml binding map

```
DEF_LIST NAME(#DEPTMEN_W) FIELDS(#DEPARTMENT)
TYPE(*WORKING)
```

```
DEF_LIST NAME(#DEPTDES_W) FIELDS(#DEPARTMENT #DEPTDESC)
TYPE(*WORKING)
```

```
CLR_LIST #DEPTMEN_W
```

```
CLR_LIST NAME(#DEPTDES_W)
```

Note: by default a working is defined with 50 entries, which is adequate for this application.

2. Continue working with function iiiFN07. The following needs to be added after the receive logic. Add your code **before** the comment line:

* Bind service to create HTTP response content

Write the RDMLX code to:

- a. SELECTLIST the department codes from list #DEPTMEN_W (the working list received from the client)
- b. FETCH the field #DEPTDESC (department description) from file DEPTAB with key #DEPARTMENT Check the I/O status of the FETCH operation. If it is not *OKAY, change the department description to the literal 'Department not found'.
- c. ADD the entry into the department description working list #DEPTDES_W
- d. ENDSELECT

Your RDMLX code might appear as follows:

```

SELECTLIST NAMED(#DEPTMEN_W)
*****
FETCH FIELDS(#DEPTDESC) FROM_FILE(DEPTAB) WITH_KEY(#DEPTMEN_W)
IF_STATUS IS_NOT(*OKAY)
CHANGE FIELD(#DEPTDESC) TO('DEPARTMENT NOT FOUND')
ENDIF
ADD_ENTRY TO_LIST(#DEPTDESC_W)
ENDSELECT
*****

```

3. Write the RDMLX to reverse the contents of #STD_TEXT using the REVERSE Built-In Function. Use the Trim intrinsic function to remove any leading or trailing blank spaces in the text.

Your RDMLX code might appear as follows:

```

***** REVERSE THE STRING
#STD_TEXTS := #STD_TEXTS.Reverse.Trim
*****

```

4. In your Studio project, expand the folder *XML List Response / Samples / RDMLX* and double click on the file `SAMPLE_RDMLX_OUTBOUND_HTTP.TXT` to open it in the test editor.
5. Select the highlighted code:

```

LANSA Integrator Text Editor - ( SAMPLE_RDMLX_OUTBOUND_HTTP.TXT )
File Edit Encoding Help
[Icons]

* Bind service to create HTTP request content
CHANGE      FIELD(#JSMXCMD) TO('BIND SERVICE(IIIPRO05_RESPONSE) TYPE(*OUTBOUND)')
USE         BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE     SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* Set fragment - DEPARTMENTLISTRESPONSE
CHANGE      FIELD(#JSMXCMD) TO('SET FRAGMENT(DEPARTMENTLISTRESPONSE) SERVICE_EXCHANGE(*FIELD)')
USE         BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE     SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* Set list - DEPARTMENT
CHANGE      FIELD(#JSMXCMD) TO('SET LIST(DEPARTMENT)')
USE         BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG #DEPARTMENT)
EXECUTE     SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* Write content
CHANGE      FIELD(#JSMXCMD) TO('WRITE INDENT(*YES) BINDTRACE(*YES)')
USE         BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE     SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* Close binding

```

6. Replace the following code in function iiiFN07 with the code above:

```

* Bind service to create HTTP response content
CHANGE FIELD(#JSMXCMD) TO('BIND SERVICE(
<<<outbound.class>>> ) TYPE(*OUTBOUND)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)

```

7. In the added code *Replace* list name DEPARTMENT with DEPTDES_W. Replace whole words only.

8. In the CHECK subroutine, add an ABORT command to the IF..ENDIF statement so that the program ends if an error has occurred.

9. Compile the function. If you are using an IBM i JSM Server, check it in and compile on the IBM i.

Your finished RDMLX code might appear as follows:

```

FUNCTION OPTIONS(*DIRECT)
* The following fields are used by the xml binding map
* #DEPARTMENT

```

* #STD_TEXTS

* The following fragments are used by the xml binding map

GROUP_BY NAME(#LISTREQ) FIELDS(#STD_TEXTS)

* The following lists are used by the xml binding map

DEF_LIST NAME(#DEPTMEN_W) FIELDS(#DEPARTMENT)

TYPE(*WORKING)

DEF_LIST NAME(#DEPTDES_W) FIELDS(#DEPARTMENT #DEPTDESC)

TYPE(*WORKING)

CLR_LIST NAMED(#DEPTMEN_W)

CLR_LIST NAMED(#DEPTDES_W)

* Open service

USE BUILTIN(JSMX_OPEN) TO_GET(#JSMXSTS #JSMXMSG
#JSMXHDL1)

EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)

* Load service

CHANGE FIELD(#JSMXCMD) TO('SERVICE_LOAD
SERVICE(HTTPInboundXMLBindService) SERVICE_CONTENT(*HTTP)
TRACE(*YES)')

USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDL1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)

EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)

* Bind service to read HTTP request content

CHANGE FIELD(#JSMXCMD) TO('BIND
SERVICE(IIIPRO05_REQUEST) TYPE(*INBOUND)
BINDTRACE(*YES)')

USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDL1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)

EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)

* Get fragment - DEPARTMENTLISTREQUEST

CHANGE FIELD(#JSMXCMD) TO('GET
FRAGMENT(DEPARTMENTLISTREQUEST)
SERVICE_EXCHANGE(*FIELD)')

```
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)
```

```
* Get list - DEPARTMENT
```

```
CHANGE FIELD(#JSMXCMD) TO('GET LIST(DEPARTMENT)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG #DEPTMEN_W)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)
SELECTLIST NAMED(#DEPTMEN_W)
```

```
*
```

```
FETCH FIELDS(#DEPTDESC) FROM_FILE(DEPTAB)
WITH_KEY(#DEPARTMENT)
IF_STATUS IS_NOT(*OKAY)
CHANGE FIELD(#DEPTDESC) TO('DEPARTMENT NOT FOUND')
ENDIF
ADD_ENTRY TO_LIST(#DEPTDES_W)
ENDSELECT
```

```
*
```

```
* REVERSE THE STRING
```

```
#STD_TEXTS := #STD_TEXTS.Reverse.Trim
```

```
*
```

```
* <<< Outbound binding logic goes here >>>
```

```
*
```

```
* Bind service to create HTTP request content
```

```
CHANGE FIELD(#JSMXCMD) TO('BIND
SERVICE(IIPRO05_RESPONSE) TYPE(*OUTBOUND)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)
```

```
* Set fragment - DEPARTMENTLISTRESPONSE
```

```
CHANGE FIELD(#JSMXCMD) TO('SET
FRAGMENT(DEPARTMENTLISTRESPONSE)
SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
```

```
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Set list - DEPARTMENT

```
CHANGE FIELD(#JSMXCMD) TO('SET LIST(DEPARTMENT)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDL1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG #DEPTDES_W)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Write content

```
CHANGE FIELD(#JSMXCMD) TO('WRITE INDENT(*YES)
BINDTRACE(*YES)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDL1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Send HTTP response content

```
CHANGE FIELD(#JSMXCMD) TO('SEND')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDL1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Close service

```
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDL1)
TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Check routine

```
SUBROUTINE NAME(CHECK) PARMS((#JSMXSTS *RECEIVED)
(#JSMXMSG *RECEIVED))
IF COND('#JSMXSTS *NE OK')
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDL1)
TO_GET(#JSMXSTS #JSMXMSG)
ABORT
```

ENDIF
ENDROUTINE

Step 8. Update Configuration Table for iiiFN07

In this step, you will use Integrator Studio to update the JSMDirect Configuration Table.

1. On the *Studio Workspace* tab, expand the *Services* node. Double click on the server you are using to open the JSMDirect *Services* dialog. This will list the services which have already been defined. Note your entries for iiiFN05_SERVICE and iiiFN06_SERVICE.
2. Right click anywhere on the white space and select *New* to open the *Direct Service Editor*.
3. Add a new entry with the following values and click *OK*. File DC@W29 on the server for your JSM instance is updated.

Field Name Value

Service	IIIFN07_SERVICE
Host	*DEFAULT
Process	IIIPRO05
Function	IIIFN07
Partition	DEM
Language	ENG (for non multilingual partitions leave blank)
Program	Leave blank
WSDL Path	Leave blank

4. Close the *Direct Services* editor.

Step 9. Create User Agent file for iiiFN07

In this step, you will create a new User Agent configuration file to test your server function. As in the previous exercise, the User Agent will act as a client querying the server function you have created. This time, however, the User Agent will invoke the new service implemented by function iiiFN07.

1. You need to copy the `iiiPRO04_client_test.lih` file to create `iiiPRO05_client_test.lih`. There are a number of ways to achieve this. This time follow these steps:
 - a. In your iii Training project, expand the folder *XML Request* and select `iiiPRO04_client_test.lih`. Open it for editing by right-clicking and selecting *Open With, Properties Editor*.
 - b. Select *Save As* from the File menu, or select the  Save As button from the toolbar.
 - c. Select the folder **XML List Request** and enter **iiiPRO05_client_test.lih** and click *Save*. Click *Yes* when prompted. Now you have a new User Agent configuration file to use for this exercise.
2. You need to modify the User Agent file to connect to the new service you have created, instead of your `iiiFN05` service from INT004. The `iiiPRO05_client_test.lih` should still be open. Make the following change:

Old Line

```
Name=iiiFN06_SERVICE  
xml.uri=/cgi-bin/jsmdirect?  
iiiFN06_SERVICE
```

New Line

```
Name=iiiFN07_SERVICE  
xml.uri=/cgi-bin/jsmdirect?  
iiiFN07_SERVICE
```

If you are using a local Windows web server and JSM Server your code should look like the following:

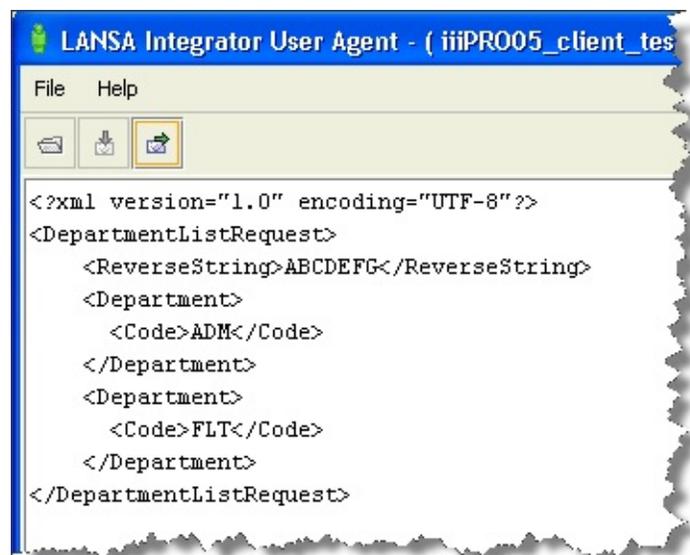
```
xml.uri=/cgi-bin/jsmdirect.exe?iiiFN07_SERVICE
```

3. The User Agent file is now ready to test `iiiFN07_SERVICE` and `iiiFN07`. Save and close the host configuration file **iiiPRO05_client_test.lih**.

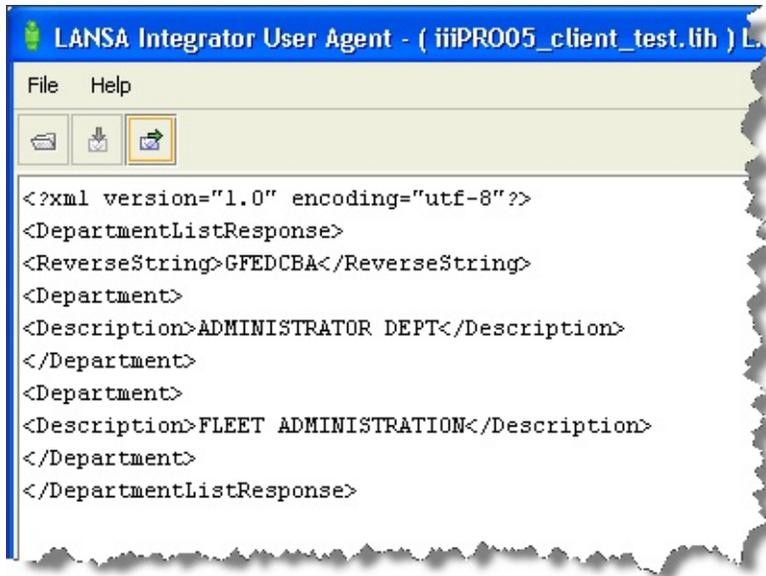
Step 10. Test iiiFN07 function with User Agent

In this step, you will test your Department List Server by running the User Agent file you created in Step 9.

1. You will run the User Agent using iiiPRO05_client_test.lih and send iiiPRO05_request.xml to the iiiFN07_SERVICE service on the JSM server.
 - a. In the *XML List Request* folder, double click on iiiPRO05_client_test.lih to open the User Agent. Click File and select the Open Source option.
 - b. Double click on iiiPRO05_request.xml. Review the XML in the *Source* tab:



- c. From the File menu select Send. Click Yes, then OK to confirm the completed transaction. The XML source has been sent to function iiiFN07 on the server.
2. Verify that the *Response* tab looks like the following:



The iiiFN07 function, called by the **iiiFN07_SERVICE** service, has processed the iiiPRO05_request.xml and returned an appropriately populated version of iiiPRO05_response.xml.

3. As in exercise INT006, you might want to try other valid Department codes. To do this, open iiiPRO05_request.xml and change the ADM or FLT text to the desired test data. The Department Descriptions in the Response will change accordingly. Invalid Department codes should return a Department Description of DEPARTMENT NOT FOUND.

Summary

Important Observations

- When you mark a tag as a list, the child tags are automatically added to the List.
- There are several ways to create XML that achieves the same functionality.

Tips & Techniques

- Mark the list before you add other child tags to the function, otherwise, the list tags will also be added to the function.

What I Should Know

- How to use the XML Bind Wizard to bind XML that contains a list.
- How to GET and SET lists using the JSM.

INT008 - Department & Employee Server

Objectives

- To create a server-side function that, given a department code, responds with a list of all employees in that department, including a list of skills for each listed employee.
- To create a User Agent file to test the server function.
- To use the *XML Bind Wizard* to create a simple mapping for the inbound request fragment.
- To use the *XML Bind Wizard* to create a more complex mapping for the outbound response, including multiple fragments and a list.
- To define outbound XML fragments dynamically within the RDMLX logic, allowing a single fragment to be written multiple times.

To achieve these objectives, you will complete the following:

- [Concepts](#)
- [Step 1. Create XML iiiPRO06_request.xml](#)
- [Step 2. Create XML iiiPRO06_response.xml](#)
- [Step 3. Create iiiFN08 Server Inbound \(request\) binding](#)
- [Step 4. Create iiiFN08 Server Outbound \(response\) binding](#)
- [Step 5. Binding Deployment & Configuration](#)
- [Step 6. Code iiiFN08 Server GET Functionality](#)
- [Step 7. Code iiiFN08 Server Database Logic and SET functionality](#)
- [Step 8. Update JSMDirect Configuration Table for iiiFN08](#)
- [Step 9. Create User Agent Host Configuration file for iiiFN08](#)
- [Step 10. Test iiiFN08 Server Function with User Agent Client](#)
- [Summary](#)

Concepts

So far in these tutorials you have learned how to send lists of items. Using fragments of XML enables you to send lists where each list item can contain repetitive information. In this exercise, for a requested department code, the server function will return a list of Employees for that department but also for each Employee the server function will return the list of skills.

- In this exercise, the server will GET the Department Code, select all the employees for the nominated department, SET the Employee's information into a fragment for each employee, and SET a list of that employee's skills. The response XML will therefore become a repeated set of Employee fragments, each with a list of skills.

This table summarizes the relationship between the input XML, the style sheets, functions and JSM command used for this exercise:

Input XML	Binding	JAR file	JSM Command
iiiPRO06_request.xml	Inbound	iiiPRO06_request.jar	GET
iiiPRO06_response.xml	Outbound	iiiPRO06_response.jar	SET

Step 1. Create XML **iiiPRO06_request.xml**

In this step, you will create the XML to SEND the Department Code from client to the server function. You can use any text editor to write XML. The required XML for the Department Code request is the same as the XML used in INT004 so you may simply copy this XML.

1. With your **iii Training** project open in Studio, select *Solutions* and use the right mouse menu to create a *New Solution* Group. Create group **Server XML Employees Request**.
2. Expand the solution group *XML Request* and open **iiiPRO04_request.xml** in the *XML Editor*.

The XML should appear something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<DepartmentRequest>
  <DepartmentCode>ADM</DepartmentCode>
</DepartmentRequest>
```

3. Select the Save As option and specify these save details:

Save in locate the **iii Training** directory, by default C:\Program Files\LANSA\Integrator\Studio\workspace\iii Training\solutions\Server XML Employees Request

File Name **iiiPRO06_request.xml**

4. Close the *XML Editor*.

Step 2. Create XML iiiPRO06_response.xml

In this step, you will create the response XML to be sent from the server and received by the User Agent. The XML will represent of a list of employees with their Employee Numbers, Given Names, and Surnames. There will be a list of skills for each employee.

1. In your **iii Training** Studio project, create a new solution group **Server XML Employees Response**.
2. Select group *Server XML Employees Response* and use the right mouse menu to create a *New Solution*
 - a. Select *XML File Editor* from the *Tool* dropdown
 - b. *Solution Group* should already contain *Server XML Employees Response*
 - c. Enter File name *iiiPRO06_response*
 - d. Copy the following to replace the default code in the editor.

```
<?xml version="1.0" encoding="UTF-8"?>
<EmployeeList>
  <Employee>
    <Code>A001</Code>
    <FullName>Fred Bloggs</FullName>
    <EmployeeSkill>
      <SkillDesc>COBOL Programming</SkillDesc>
    </EmployeeSkill>
    <EmployeeSkill>
      <SkillDesc>RPG Programming</SkillDesc>
    </EmployeeSkill>
  </Employee>
  <Employee>
    <Code>A002</Code>
    <FullName>Joe Smith</FullName>
    <EmployeeSkill>
      <SkillDesc>C++ Introduction</SkillDesc>
    </EmployeeSkill>
    <EmployeeSkill>
      <SkillDesc>Help Desk Management</SkillDesc>
    </EmployeeSkill>
  </Employee>
</EmployeeList>
```

</EmployeeList>

3. Notice the structure of this XML document. It has a list of employees. Each employee has details (Code and Full Name) as well as a list of skills.
4. Save your new XML and close the text editor.

Step 3. Create iiiFN08 Server Inbound (request) binding

In this step, you will use the *XML Binding Wizard* to create the inbound request binding **iiiPRO06_request.jar**.

1. With your **iii Training** project open in Studio:
 - a. Right-click on the *Server XML Employees Request* folder and select *New Solution*.
 - b. Select the *XML Binding Wizard* from the *Tool* dropdown
 - c. *Solution Group* should already contain *Server XML Employees Request*
 - c. Type **iiiPRO06_request** in the *Solution File input* field. Click *OK*.
2. The *XML Binding Project* dialog appears. Click on the *Browse* button. Select the file **iiiPRO06_request.xml** then click on the *Open* button. On the *New XML Binding Project* dialog, click *OK*. The *XML Binding Wizard* main window will now appear. Change *Package* to **com.iiicompany.service.xml**.
3. Click the *Inbound* checkbox. Select the newly-created *Inbound* tab.
4. Map the XML tag **DepartmentCode** to the LANSAs field **DEPARTMENT** as follows:
 - a. From the *Inbound* tab, mark the **DepartmentRequest** root tag as a *Fragment*.
 - b. Select the **Department Code** tag and select *Edit* from the context menu. Enter **DEPARTMENT** for the *Name* and click *OK*

Your *XML Binding* should look like the following:



5. Build your service by clicking the  *Build* button on the toolbar.
Your *JAR* file has now been generated and compiled.

6. *Save* your work and *Exit* the *XML Binding Wizard*.

Step 4. Create iiiFN08 Server Outbound (response) binding

In this step, you will use the *XML Bind Wizard* to create the outbound response binding **iiiPRO06_response.jar** on the server. In this binding, the list of Employee Skills will be represented with a list, but the list of Employees will be defined as a fragment, in order to avoid nested lists. The Employee fragment will be written multiple times in the RDMLX code.

1. First, create a new XML Binding Wizard solution in your Server XML Response group:
 - a. With the **iii Training** project open in Studio, right-click on the *Server XML Employee Response* solution group and select *New Solution*.
 - b. The *Server XML Employee Response* group should be pre-selected. Select the *XML Binding Wizard*. Enter **iiiPRO06_response** in the *File* input field. Click *OK*.
2. The *XML Binding Project* panel appears. Click the *Browse* button. Select the **iiiPRO06_response.xml** file and then click the *Open* button, then *OK on the New XML Binding Project dialog*.

The *XML Binding Wizard* main window will now appear. Change *Package* to **com.iiicompany.service.xml**,

3. Click the Outbound checkbox. Select the newly-created Outbound tab.
4. The bindings for the response XML are more complex than the inbound.
 - The *EmployeeList* root tag and the *Employee* tag need to be marked as *fragments*
 - *EmployeeSkill* needs to be marked as a *list*
 - *Code*, *FullName*, and *SkillDesc* need to be mapped to specific LANSAs *fields*.

The Employee fragment will be written repeatedly in the RDMLX code.

To do this, map the Employee details as follows:

- a. Right click on the *EmployeeList* root tag. Select *Fragment* from the pop-up menu.
- b. Right click on the *Employee* tag. Select *Fragment* from the pop-up menu.
- c. Right click on the *Code* tag and select *Edit*. Enter **EMPNO** for the Name and click *OK*.

- d. Right click on the FullName tag and select Edit. Type **FULLNAME** for the Name and click OK.
- e. Right click on either of the EmployeeSkill tags and select List.
- f. Expand the EmployeeSkill tags to see their sub-tags. Right click or double click on either of the SkillDesc tags to edit them. Enter **SKILDESC** for the Name and click OK.

Your *XML Binding* should look like the following:



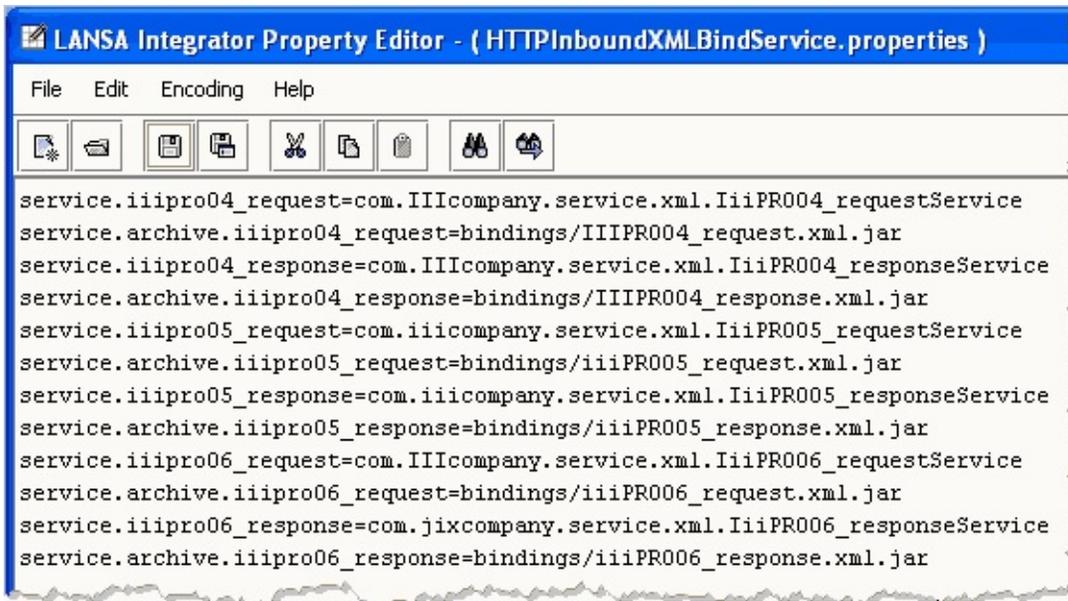
5. Your *XML Binding* definitions are complete. Now, generate and compile the outbound response service. Select the Project menu and choose the Build option.
6. Save your work and *Exit* the *XML Binding Wizard*.

Step 5. Binding Deployment & Configuration

In this step, you will deploy the jar files and update the HTTPInboundXMLBindService properties file using the Integrator Studio. With your **iii Training** project open in Studio and with the JSM Server instance open (right hand panel):

1. Retrieve the HTTPInboundXMLBindService.properties file.
 - a. Open the *properties* folder in the right panel (i.e. in the JSM server instance).
 - b. Scroll down and locate the HTTPInboundXMLBindService.properties file. Right click on the file and select the *Retrieve Section* option in the pop-up menu. A message will be displayed, "Replace HTTPInboundXMLBindService.properties" because there is already an existing copy in your local folder. Click *Yes to overwrite it*.
 - c. Open the Integrator / *properties* folder in the left panel (i.e. your local directory). Right click on the HTTPInboundXMLBindService.properties file and select the *Open* option from the pop-up menu to open the *Properties Editor (1)*. This file has been updated from the server and is ready for you to modify.
 - d. As in the previous exercise:
 - Open the HTTPInboundXMLBindService.properties in the *Server XML Employees Request* folder, and copy the code into the *Properties Editor (1)*. Close the second editor
 - Open the HTTPInboundXMLBindService.properties in *Server XML Employees Response* folder and copy this code to *Properties Editor (1)*. Close the second editor.

Your properties file should now look like the following:



- f. Save your changes and exit the *Properties editor*.
 - g. Right click on your local `HTTPInboundXMLBindService.properties` file (in the left panel, in *Integrator / Properties*) and select the *Publish Section* option from the pop-up menu. A message will ask you to confirm publish. Select *Yes*.
2. Deploy the jar files.
- You must copy the jar files from your PC into the appropriate JSM server directory. You will send the jar files to the directory `\instance\bindings`.
- a. In the *Server XML Employees Response* folder, right click on `iiiPRO06_response.xml.jar` and select *Send To / Integrator Folder* from the pop-up menu. Select *Yes* when asked if you want to copy the file over.
 - b. Repeat for `iiiPRO06_request.xml.jar`. Alternatively you could drag-and-drop the jar files into the *Integrator / bindings* folder.
 - c. In the left panel, expand the *Integrator / bindings* folder. One at a time, right click on each jar file and select *Publish File*. Select *Yes* when prompted.

Step 6. Code **iiiFN08** Server GET Functionality

In this step, you will begin to write the RDMLX for the Server side GET. The server function will receive the XML from the client.

1. Create a new LANSAPROCESS named **iiiPRO06 XML Employees Server**, where **iii** is your unique 3 characters. (If the process already exists, select a different set of characters for **iii**.)
2. Create a new function named **iiiFN08 Department & Employees Server**, belonging to process **iiiPRO06**. Create the function without using a template and make sure that Enabled for RDMLX is checked.
3. With your **iii Training** project open in Studio, expand the *Solutions / Server XML Employees Request / samples / RDMLX* folder and open file **SAMPLE_RDMLX_INBOUND_HTTP.txt** in the *Text Editor*. Copy all the code into function **iiiFN08**, replacing existing code.
4. Change the GROUP_BY name to DEPTREQ. This GROUP_BY is a reference for mapping field DEPARTMENT. The code generator does not currently recognize RDMLX name limits.
5. At this point your function has all the logic needed to receive the department code request XML
6. Save this function. It is not yet complete and will be finished in a later step.

Step 7. Code iiiFN08 Server Database Logic and SET functionality

In this step, you will write the RDMLX to SELECT using the logical view PSLMST1 (Personnel by Department) to get a list of employees, then SELECT over the file PLSKL (Personnel Skills) to get a list of that employee's skills. The RDMLX for the server side SET will be performed inline during this internal logic. This is different from previous exercises, where the internal logic and SET commands were separated sequentially.

In this exercise, you will not be performing each of these commands in sequence, after the internal logic is finished. Instead, the outbound BIND and root fragment SET will come immediately after the inbound BIND and GET commands (before any database accesses). Two more SET commands will go into the outer SELECT statement. Finally, the WRITE and SEND commands will follow the last ENDSELECT.

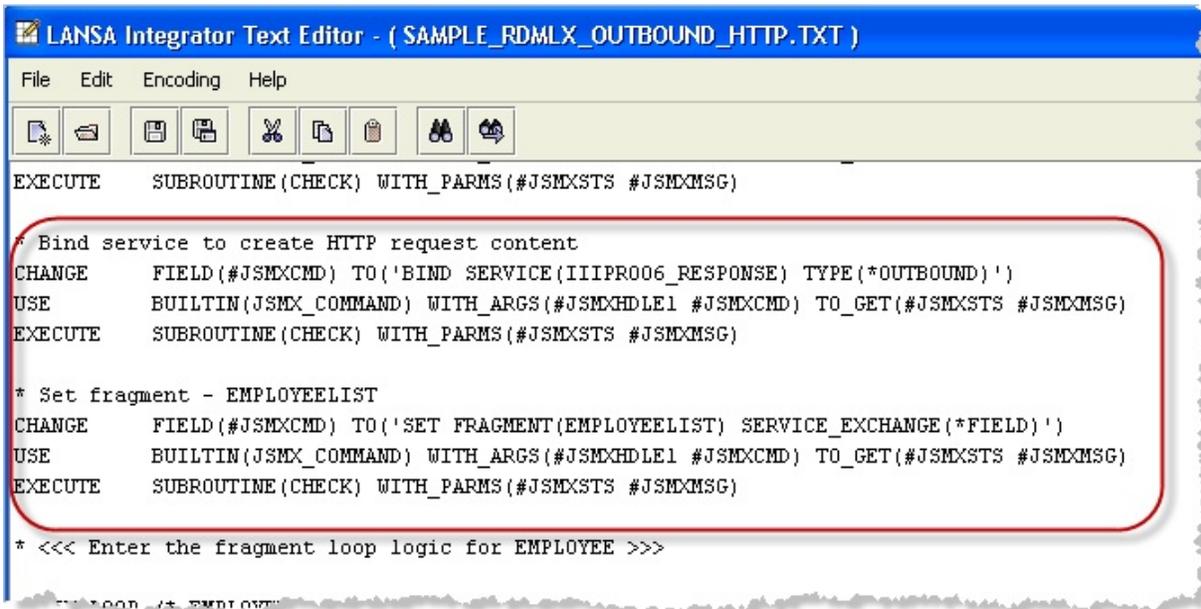
The overall structure of the iiiFN08 function is as follows:

1. **JSMX_OPEN**
2. SERVICE_LOAD (HTTPInboundXMLBindService)
3. BIND (inbound)
4. GET (department code fragment)
5. BIND (outbound)
6. SET (employee list fragment)
7. SELECT over PSLMST1
 - a. SELECT over PLSKL
 - b. FETCH from SKLTAB
 - c. ENDSELECT
 - d. SET (employee fragment)
 - e. SET (employee skill list)
8. ENDSELECT
9. WRITE
10. SEND
11. **JSMX_CLOSE**

1. In your function iiiFN08 locate the following code and delete it:

```
* Bind service to create HTTP response content
CHANGE FIELD(#JSMXCMD) TO('BIND SERVICE(
<<<outbound.class>>> ) TYPE(*OUTBOUND)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

2. With your **iii Training** project open in Studio, in the folder *Server XML Employees Response /samples / RDMLX* open the file **SAMPLE_RDMLX_OUTBOUND_HTTP.txt** in the *Text Editor*. Locate the highlighted code:



Copy and paste the highlighted code into your function, to replace the deleted code block.

3. Add a working list definition named EMPWORK to the top of your function containing field SKILDESC. This is the list of skills for each employee, which is to be returned. Your code should look like the following:

```
DEF_LIST NAME(#EMPWORK) FIELDS(#SKILDESC)
TYPE(*WORKING) ENTRYS(*MAX)
```

4. After the outbound BIND and SET logic just added, write the RDMLX code to:

- a. SELECT employee numbers and names from the logical file PSLMST1, using the DEPARTMENT key field.
- b. Inside the SELECT, set up the field Fullname from the Given Name and Surname fields.
- c. Still within that SELECT loop, CLR_LIST EMPWORK, then SELECT over PLSKL to get a list of skill codes for each employee, using the employee number.
- d. Inside the inner (nested) SELECT, FETCH the skill description from SKLTAB for each skill code.
- e. ADD the entry into the employee skills working list EMPWORK.
- f. End both SELECTS.

The RDMLX code might appear as follows:

```

SELECT FIELDS(#EMPNO #SURNAME #GIVENAME) FROM_FILE(PSLMST1)
#FULLNAME := #GIVENAME + ' ' + #SURNAME
CLR_LIST NAMED(#EMPWORK)
SELECT FIELDS(#SKILCODE) FROM_FILE(PLSKL) WITH_KEY(#EMPNO)
FETCH FIELDS(#SKILDESC) FROM_FILE(SKLTAB) WITH_KEY(#SKILCODE)
ADD_ENTRY TO_LIST(#EMPWORK)
ENDSELECT
* Set employee fragment
* Set employeeskills list
ENDSELECT

```

Note: Comments have been added to this code for clarity in the next steps.

5. As before, you will complete the function by using code generated for you in the **SAMPLE_RDMLX_OUTBOUND_HTTP.txt** file. Locate the highlighted code:

```

LANSA Integrator Text Editor - ( SAMPLE_RDMLX_OUTBOUND_HTTP.TXT )
File Edit Encoding Help
* <<< Enter the fragment loop logic for EMPLOYEE >>>
BEGIN_LOOP /* EMPLOYEE */
* Set fragment - EMPLOYEE
CHANGE FIELD(#JSMXCMD) TO('SET FRAGMENT(EMPLOYEE) SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)
* Set list - EMPLOYEEESKILL
CHANGE FIELD(#JSMXCMD) TO('SET LIST(EMPLOYEEESKILL)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG #EMPLOYEEESKILL)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

```

6. Copy the highlighted code shown above into your function to replace the comments included in step 4f .

- * Set employee fragment
- * Set employeeskills list

7. Change the working list name in the TO_GET() keyword to refer to your list #EMPWORK. Your code should look like the following:

```

*set list - EMPLOYEEESKILL
CHANGE FIELD(#JSMXCMD) TO('SET LIST(EMPLOYEEESKILL)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG #EMPWORK)

```

8. Review your function's RDMLX code following the last ENDSELECT. You will find that the generated code which you copied initially, already contains the required WRITE, SEND and CLOSE logic.

9. In the CHECK subroutine, add an ABORT command to the IF..ENDIF statement so that the program ends if an error has occurred.

10. Save and compile the function. If you are using an IBM i JSM Server, check it into the IBM i and compile it.

Your finished RDMLX code might appear as follows:

```

FUNCTION OPTIONS(*DIRECT)
* The following fields are used by the xml binding map
* #DEPARTMENT
* The following fragments are used by the xml binding map
GROUP_BY NAME(#DEPTREQ) FIELDS(#DEPARTMENT)

```

*

```
DEF_LIST NAME(#empwork) FIELDS(#skildesc) ENTRIES(200)
```

* Open service

```
USE BUILTIN(JSMX_OPEN) TO_GET(#JSMXSTS #JSMXMSG  
#JSMXHDLE1)  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS  
#JSMXMSG)
```

* Load service

```
CHANGE FIELD(#JSMXCMD) TO('SERVICE_LOAD  
SERVICE(HTTPInboundXMLBindService) SERVICE_CONTENT(*HTTP)  
TRACE(*YES)')  
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1  
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS  
#JSMXMSG)
```

* Bind service to read HTTP request content

```
CHANGE FIELD(#JSMXCMD) TO('BIND  
SERVICE(IIIPRO06_REQUEST) TYPE(*INBOUND)  
BINDTRACE(*YES)')  
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1  
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS  
#JSMXMSG)
```

* Get fragment - DEPARTMENTREQUEST

```
CHANGE FIELD(#JSMXCMD) TO('GET  
FRAGMENT(DEPARTMENTREQUEST)  
SERVICE_EXCHANGE(*FIELD)')  
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1  
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS  
#JSMXMSG)
```

* Bind service to create HTTP request content

```
CHANGE FIELD(#JSMXCMD) TO('BIND  
SERVICE(IIIPRO06_RESPONSE) TYPE(*OUTBOUND)')
```

```
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Set fragment - EMPLOYEELIST

```
CHANGE FIELD(#JSMXCMD) TO('SET FRAGMENT(EMPLOYEELIST)
SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

```
SELECT FIELDS(#EMPNO #SURNAME #GIVENAME)
FROM_FILE(PSLMST1) WITH_KEY(#DEPTMENT)
#FULLNAME := #GIVENAME + ' ' + #SURNAME
CLR_LIST NAMED(#EMPWORK)
SELECT FIELDS(#SKILCODE) FROM_FILE(PSLSKL)
WITH_KEY(#EMPNO)
FETCH FIELDS(#SKILDESC) FROM_FILE(SKLTAB)
WITH_KEY(#SKILCODE)
ADD_ENTRY TO_LIST(#EMPWORK)
ENDSELECT
```

* Set fragment - EMPLOYEE

```
CHANGE FIELD(#JSMXCMD) TO('SET FRAGMENT(EMPLOYEE)
SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Set list - EMPLOYEEESKILL

```
CHANGE FIELD(#JSMXCMD) TO('SET LIST(EMPLOYEEESKILL)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG #EMPWORK)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
ENDSELECT
```

* Write content

```
CHANGE FIELD(#JSMXCMD) TO('WRITE INDENT(*YES)
BINDTRACE(*YES)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Send HTTP response content

```
CHANGE FIELD(#JSMXCMD) TO('SEND')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Close service

```
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1)
TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Check routine

```
SUBROUTINE NAME(CHECK) PARMS((#JSMXSTS *RECEIVED)
(#JSMXMSG *RECEIVED))
IF COND(#JSMXSTS *NE OK')
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1)
TO_GET(#JSMXSTS #JSMXMSG)
ENDIF
ENDROUTINE
```

Step 8. Update JSMDirect Configuration Table for iiiFN08

In this step, you will use the Integrator Studio to update the JSMDirect Configuration Table.

1. Click on the *Studio Workspace*. Expand the *Service* node. You will see the service you created before.
2. Double click on the service to open it. The Direct Services tab will appear showing the entries you have created in previous exercises.
3. Add a new entry with the following values:

Field Name	Value
-------------------	--------------

Service	iiiFN08_SERVICE
---------	-----------------

Host	*DEFAULT
------	----------

Process	iiiPRO06
---------	----------

Function	iiiFN08
----------	---------

Partition	DEM
-----------	-----

Language	ENG (for non multilingual partitions leave blank)
----------	---

Program	Leave blank
---------	-------------

WSDL	Leave blank
------	-------------

4. Close the *Direct Services* tab.

Step 9. Create User Agent Host Configuration file for iiiFN08

In this step, you will create a new User Agent configuration file to test your server function. As in previous exercises, the User Agent will act as a client querying the server you have created.

1. Copy the **iiiPRO05_client_test.lih** file to create **iiiPRO06_client_test.lih**:
 - a. In your **XML List Request** folder, select **iiiPRO05_client_test.lih**. Open it for editing by right-clicking and selecting Open With / Properties Editor. Click Save As, or select Save As from the File menu. Save it to folder **Server XML Employees Request**.
 - b. Enter **iiiPRO06_client_test.lih** and click Save. Click Yes when prompted. Now you have a new User Agent file to use for this exercise.
2. You need to modify the User Agent file to connect to the new service you have created, instead of your old service. The **iiiPRO06_client_test.lih** should still be open. Make the following change:

Old Line

name=iiiFN07_SERVICE
xml.uri=/cgi-bin/jsmdirect?
iiiFN07_SERVICE

New Line

name=iiiFN08_SERVICE
xml.uri=/cgi-bin/jsmdirect?
iiiFN08_SERVICE

If your using a local Windows web server and JSM Server the uri should look like the following:

xml.uri=/cgi-bin/jsmdirect.exe?iiiFN08_SERVICE

3. The User Agent Host Configuration file is now ready to test iiiFN08_SERVICE and iiiFN08. Save and close **iiiPRO06_client_test.lih**.

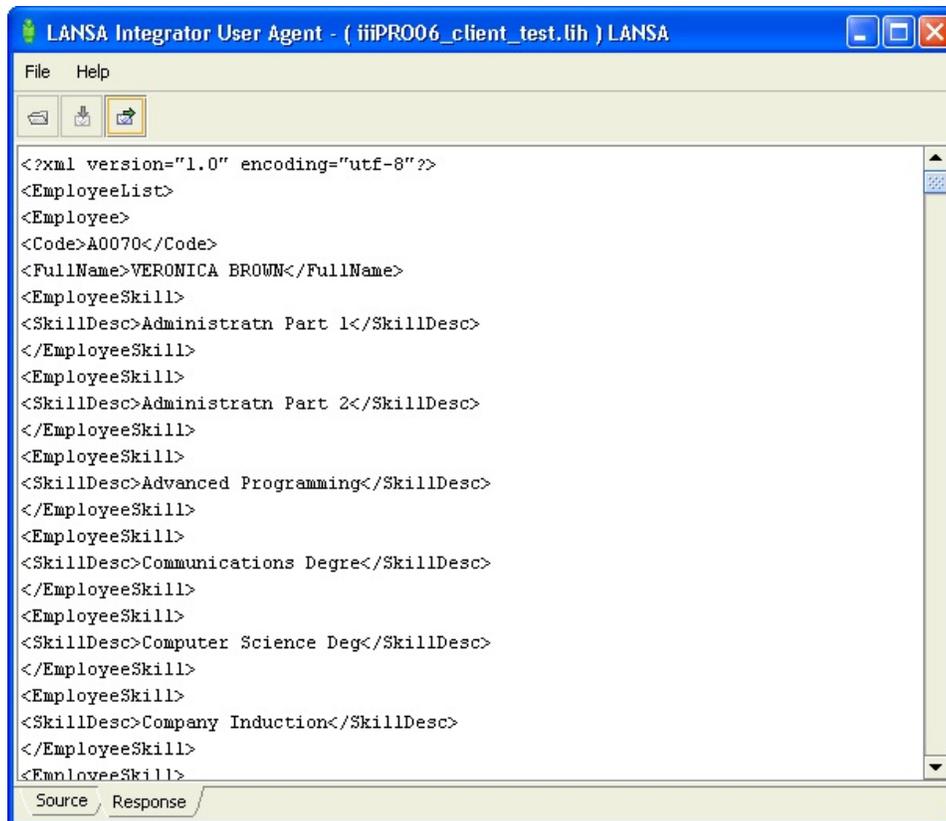
Step 10. Test iiiFN08 Server Function with User Agent Client

In this step, you will test your Department & Employee Server by running the User Agent using the host configuration file you created in Step 9.

1. In the folder *Server XML Employees Request* open the User Agent using **iiiPRO06_client_test.lih** and send iiiPRO06_request.xml to the **iiiFN08_SERVICE** service on the JSM server.
 - a. Double click iiiPRO06_client_test.lih to run the User Agent. Click File and select Open Source.
 - b. Double click on iiiPRO06_request.xml. Review the XML in the *Source* tab:



- c. Click File and select Send. Click Yes, then OK to confirm the completed transaction. The XML source has been sent to the server.
2. Verify that the server has sent the correct list of Employees and Employee Skills. The Response tab should have a large amount of text in XML format—each department has many employees, and each employee has many skills. Browse the response XML and check that it is correct.



The screenshot shows a window titled "LANSA Integrator User Agent - (iiiPRO06_client_test.lrh) LANSA". The window contains a text area with the following XML content:

```
<?xml version="1.0" encoding="utf-8"?>
<EmployeeList>
<Employee>
<Code>A0070</Code>
<FullName>VERONICA BROWN</FullName>
<EmployeeSkill>
<SkillDesc>Administratrn Part 1</SkillDesc>
</EmployeeSkill>
<EmployeeSkill>
<SkillDesc>Administratrn Part 2</SkillDesc>
</EmployeeSkill>
<EmployeeSkill>
<SkillDesc>Advanced Programming</SkillDesc>
</EmployeeSkill>
<EmployeeSkill>
<SkillDesc>Communications Degre</SkillDesc>
</EmployeeSkill>
<EmployeeSkill>
<SkillDesc>Computer Science Deg</SkillDesc>
</EmployeeSkill>
<EmployeeSkill>
<SkillDesc>Company Induction</SkillDesc>
</EmployeeSkill>
<EmnloyeeSkill>
```

At the bottom of the window, there are two tabs: "Source" and "Response".

The iiiFN08 function, called by the **iiiFN08_SERVICE** service, has processed the iiiPRO06_request.xml and returned an appropriately populated version of iiiPRO06_response.xml.

3. You might want to try other valid Department codes. To do this, open iiiPRO06_request.xml and change the text to FLT or other desired test data. The Department Descriptions in the Response will change accordingly. Invalid Department codes should return a blank list.

Summary

Important Observations

- Use repeated fragments whenever the XML structure involves multiple occurrences of structure. For example, use repetitive fragments for lists within a list.
- JSMX commands were performed in parallel to database access RDMLX. This is a useful technique for building complex XML.

Tips & Techniques

- Multiple occurrences of the same fragment can mimic the behavior of lists. This is an effective technique for creating nested lists.
- Looping over the JSMX "SET" command is useful when implementing nested lists.

What I Should Know

- How to use repeated fragments to create nested lists in XML
- How to integrate JSMX commands into RDMLX code
- How to build server-side functions that create complex response XML

INT009 – Department & Employee Client (Optional)

Objectives:

- To learn how to create a client request RDMLX function to test your Department & Employee Server.
- To create a wrapper RDML function that calls your client function from a Windows user interface.
- To learn how to integrate client- and server-side functions with HTTPInboundXMLBindService and HTTPOutboundXMLBindService.
- To see the overall structure of the client/server transaction.

To achieve these objectives, you will complete the following:

- [Concepts](#)
- [Step 1. Create iiiFN09 Client Outbound \(request\) bindings](#)
- [Step 2. Create iiiFN09 Client Inbound \(response\) bindings](#)
- [Step 3. Binding Deployment and Configuration](#)
- [Step 4. Code iiiFN10 Client Wrapper Function](#)
- [Step 5. Code iiiFN09 Client SET functionality](#)
- [Step 6. Code iiiFN09 Client GET functionality](#)
- [Step 7. Test iiiFN08 Server with Client functions](#)
- [Step 8. A Client Visual Form \(optional\)](#)
- [Summary](#)

Concepts

You have already written the Department & Employee Server function, and created a service interface for it, in INT008. Now you will create a client function using HTTPOutboundXMLBindService. This client function will allow the user to input a department code, create a request XML document, send the XML to your iiiFN08_SERVICE to be processed, and display the response XML in a user-friendly list.

Step 1. Create iiiFN09 Client Outbound (request) bindings

In this step, you will create a new jar file from iiiPRO06_request.xml. In INT008, your iiiPRO06_request.xml bindings were inbound. This time, however, you are creating a client function, so the bindings for your request XML will be outbound.

You are binding the same XML as in INT008 for your request and response, since you are building another part of the same interaction. The structure (XML) of the request and response messages is the same for the server and client. You are simply building the opposite side of the transaction.

1. With your **iii Training** project open in Studio, create a new *Solution Group / Client XML Employees Request*. Select this new group and use the *XML Binding Wizard*. Give the solution file the name iiiPRO07_request. *Browse* to select iiiPRO06_request.xml from folder Server XML Employees Request for the XML source. Change the *Package* name to com.iiiCompany.service.xml.
2. Click the Outbound checkbox in the *XML Binding Wizard*. Mark the root DepartmentRequest node as a fragment. Edit the DepartmentCode node and set its Name to DEPARTMENT.
3. Build and compile your solution. Save your work and close the *XML Binding Wizard*.

Step 2. Create iiiFN09 Client Inbound (response) bindings

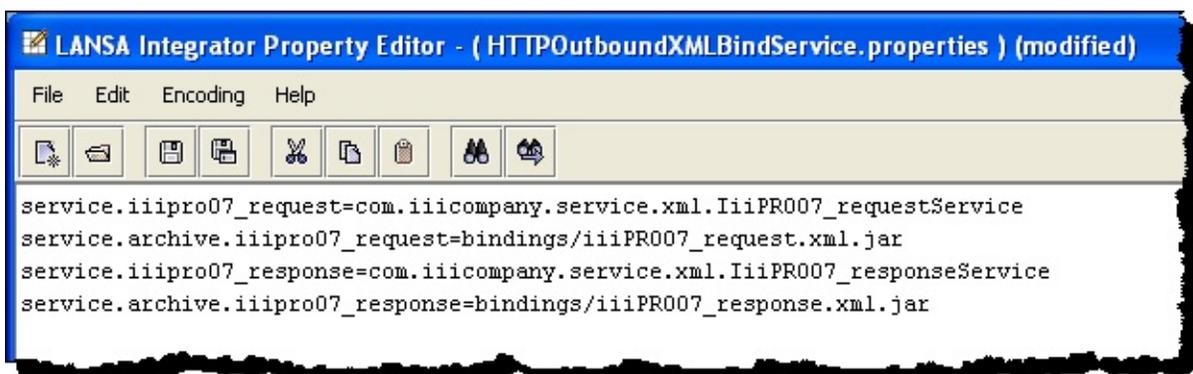
In this step, you will create bindings for iiiPRO06_response.xml. In INT008, this file had **outbound** bindings, since that function was acting as the server. This time, you are creating a client function, so the bindings for your response XML will be **inbound**. Note that you are binding the same XML as in INT008.

1. In your **iii Training** project, create a *New Solution Group / Client XML Employees Response*. Select the new group and create a *New Solution*. Use the *XML Binding Wizard* to create a new *Solution*. Give the solution file the name iiiPRO07_response. Browse to select iiiPRO06_response.xml from folder *Server XML Employees Response* for the XML source. Change the Package name to com.iiicompany.service.xml.
2. Click the Inbound checkbox in the *XML Binding Wizard*.
 - a. Mark the root EmployeeList node as a *fragment*.
 - b. Edit the Code node and set its Name to EMPNO.
 - c. Edit the FullName node and set its Name to FULLNAME.
 - d. Mark EmployeeSkill as a *list*.
 - e. Edit SkillDesc and set its Name to SKILDESC.
3. Build and compile your solution. Save your work and close the *XML Binding Wizard*.

Step 3. Binding Deployment and Configuration

In this step, you will deploy your binding jar files.

1. Send `iiiPRO07_response.jar` and `iiiPRO07_request.jar` to the Integrator folder from your workspace.
2. Publish `iiiPRO07_response.jar` and `iiiPRO07_request.jar` to the server, from the `/bindings` folder.
3. For this exercise, you will be using the `HTTPOutboundXMLBindService`. Therefore, you must define a service in the `HTTPOutboundXMLBindService.properties` file. This will be used in your client-side RDMLX function. The server will not use this service at all.
 - a. Use *Retrieve Section* on the `HTTPOutboundXMLBindService.properties` file from the server. (This action simply creates a local file for you to edit. There is no project specific data to retrieve at this time.) This step requires your server instance to be opened from your project tab.
 - b. From your project *Integrator / properties* folder open your local section of the `HTTPOutboundXMLBindService.properties` file. As in INT009, now open the properties for this **outbound** service in your new solution groups, and copy each in turn to the properties section file you just created. Your file should now look like the following:



The screenshot shows a window titled "LANSa Integrator Property Editor - (HTTPOutboundXMLBindService.properties) (modified)". The window has a menu bar with "File", "Edit", "Encoding", and "Help". Below the menu bar is a toolbar with icons for file operations. The main text area contains the following properties:

```
service.iiipro07_request=com.iiicompany.service.xml.IiiPR007_requestService
service.archive.iiipro07_request=bindings/iiiPR007_request.xml.jar
service.iiipro07_response=com.iiicompany.service.xml.IiiPR007_responseService
service.archive.iiipro07_response=bindings/iiiPR007_response.xml.jar
```

- c. Save and close the properties file.
- d. Use *Publish Section* to update the `HTTPOutboundXMLBindService.properties` file on the server.

Step 4. Code iiiFN10 Client Wrapper Function

In this step, you will create the RDML (not RDMLX) wrapper function for your client. This will allow you to interact with your server from a simple Windows program.

1. Create a new LANSAPROCESS named iiiPRO07 JSM XML Employees Client, where iii is your unique 3 characters. (If the process already exists, select a different set of characters for iii.)
2. Create a new function named iiiFN10 Display Department & Employee Client belonging to process **iiiPRO07**. Do not create the function from a template. Do not make it an RDMLX enabled function.
3. The basic structure of this program accepts Department Codes input from the user, calls RDMLX function iiiFN09 and then displays the resulting employees list. You will need a browselist and a working list to hold this data.
 - a. Define WL_EMPSKL as a working list of FULLNAME and SKILDESC.
 - b. Define BL_EMPSKL as a browselist of FULLNAME and SKILDESC.
4. Code the input, function call, and output of the display function.
 - a. Insert BEGIN_LOOP and END_LOOP commands.
 - b. Inside the loop, insert a REQUEST command to request DEPARTMENT and display the browse list BL_EMPSKL.
 - c. Clear both the working and browse list.
 - d. Exchange field DEPARTMENT.
 - e. Call function iiiFN09, passing the working list WL_EMPSKL (to allow the populated data to return).
 - f. Loop through the working list, adding entries to the browselist on each iteration.

Your finished code should look something like this:

```
FUNCTION OPTIONS(*DIRECT)
DEF_LIST NAME(#WL_EMPSKL) FIELDS(#FULLNAME #SKILDESC) T
DEF_LIST NAME(#BL_EMPSKL) FIELDS(#FULLNAME #SKILDESC) E

BEGIN_LOOP
REQUEST FIELDS(#DEPARTMENT) BROWSELIST(#BL_EMPSKL)
```

```
CLR_LIST NAMED(#WL_EMPSKL)
CLR_LIST NAMED(#BL_EMPSKL)
```

```
EXCHANGE FIELDS(#DEPARTMENT)
CALL PROCESS(*DIRECT) FUNCTION(IIIFN09) PASS_LST(#WL_EMPS
```

```
SELECTLIST NAMED(#WL_EMPSKL)
ADD_ENTRY TO_LIST(#BL_EMPSKL)
ENDSELECT
```

```
END_LOOP
```

5. Save, compile and close the function.

Step 5. Code **iiiFN09** Client SET functionality

In this step, you will begin to write the RDMLX for the **client**. The client will send a request document to the server, and receive a list of employees from the service created in INT008. For now, you will code only the creation and sending of the request XML.

The overall structure of the completed client function **iiiFN09** will be:

1. **JSMX_OPEN**
 2. SERVICE_LOAD HTTPOutboundXMLBindService
 3. BIND outbound
 4. SET DepartmentRequest fragment
 5. WRITE DepartmentRequest fragment
 6. SEND HOST() URI()
 7. BIND service IIIPRO07_SERVICE inbound
 8. GET EMPLOYEEELIST fragment
 9. BEGIN_LOOP
 - a. GET EMPLOYEE fragment
 - b. LEAVE if NOFRAGMENT
 - c. GET EMPLOYEEESKILLS list
 - d. SELECT LIST
 - e. Add entry WL_EMPSKL
 - f. End SELECT
 10. END_LOOP
 11. SERVICE_UNLOAD
 12. JSMX_CLOSE
1. Create a new function named **iiiFN09** Department & Employee Client belonging to process **iiiPRO07** where **iii** is your unique 3 characters. Create the function without using an application template. Make it an RDMLX enabled function.
 2. In your **iii Training** project, expand the *Client XML Employees Request / samples / RDMLX* folder. Copy the code from **SAMPLE_RDMLX_OUTBOUND_HTTP.TXT** to replace the existing

code in function **IIIFN09**.

3. Change the GROUP_BY DEPARTMENTREQUEST name to DEPTREQ.
The Studio code generator does not currently recognize RDMLX name limits.
This is a reference for binding field DEPTMENT.
4. You need to specify that you are receiving a working list from your wrapper RDML function.
 - a. Define working list WL_EMPSKL as it was defined in your wrapper function.
 - b. Add a RCV_LIST to the FUNCTION OPTIONS(*DIRECT) to receive working list WL_EMPSKL.
5. Complete the SEND command. The HOST keyword must point to the web server associated with your JSM Server. The example code below points to an IBM i host. If you are using a local web server and JSM server it could be defined as "localhost". The URI keyword refers to the service being called - **IIIFN08_SERVICE**. Your code should look similar to the following:

* Send HTTP request content

```
CHANGE FIELD(#JSMXCMD) TO('SEND HOST(10.44.10.236:80)
URI(/cgi-bin/jsmdirect?IIIFN08_SERVICE)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

Note: The *XMLBind Wizard* generates the above code with a **URL** keyword. This is an error and you should change this keyword to **URI** as shown in the above code.

6. Delete the BIND to the inbound.class code, shown below. The response logic will be taken from generated sample RDMLX in the next step

* Bind service to read HTTP response content

```
CHANGE FIELD(#JSMXCMD) TO('BIND SERVICE(
<<<inbound.class>>> ) TYPE(*INBOUND) BINDTRACE(*YES)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

Your RDMLX code should currently look like the following:

```

FUNCTION OPTIONS(*DIRECT) RCV_LIST(#wl_empskl)
* The following fields are used by the xml binding map
* #DEPARTMENT
* The following fragments are used by the xml binding map
GROUP_BY NAME(#DEPTREQ) FIELDS(#DEPARTMENT)
DEF_LIST NAME(#WL_EMPSKL) FIELDS(#FULLNAME #SKILDESC)
TYPE(*WORKING) ENTRIES(9999)
* Open service
USE BUILTIN(JSMX_OPEN) TO_GET(#JSMXSTS #JSMXMSG
#JSMXHDLE1)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)

* Load service
CHANGE FIELD(#JSMXCMD) TO('SERVICE_LOAD
SERVICE(HTTPOutboundXMLBindService) TRACE(*YES)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)

* Bind service to create HTTP request content
CHANGE FIELD(#JSMXCMD) TO('BIND
SERVICE(IIPRO07_REQUEST) TYPE(*OUTBOUND)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)

* Set fragment - DEPARTMENTREQUEST
CHANGE FIELD(#JSMXCMD) TO('SET
FRAGMENT(DEPARTMENTREQUEST)
SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)

* Write content
CHANGE FIELD(#JSMXCMD) TO('WRITE INDENT(*YES)
BINDTRACE(*YES)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1

```

```

#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMXSTS
#JSMXMSG)
* Close binding
CHANGE FIELD(#JSMXCMD) TO('CLOSE')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMXSTS
#JSMXMSG)
* Send HTTP request content
CHANGE FIELD(#JSMXCMD) TO('SEND HOST(10.44.10.236:80)
URI(/cgi-bin/jsmdirect?IIIFN08_SERVICE)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMXSTS
#JSMXMSG)
* <<< Inbound binding logic goes here >>>

* Unload service
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
'SERVICE_UNLOAD') TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMXSTS
#JSMXMSG)
* Close service
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1)
TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMXSTS
#JSMXMSG)
* Check routine
SUBROUTINE NAME(CHECK) PARMs((#JSMXSTS *RECEIVED)
(#JSMXMSG *RECEIVED))
IF COND('#JSMXSTS *NE OK')
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1)
TO_GET(#JSMXSTS #JSMXMSG)
MENU MSGTXT('Java service error has occurred')
ENDIF
ENDROUTINE

```

7. Save this function. **It is not yet complete. It will be finished in the next step.** At this stage, the server function has been sent the XML request document to process.

Step 6. Code iiiFN09 Client GET functionality

In this step you will use the inbound part of the RDMLX generated for the *Client XML Employees Response* solution to complete function iiiFN09.

1. With your **iii Training** project open in Studio, expand the folder *Solutions / Client XML Employees Response / sample / RDMLX* and copy the following code from *SAMPLE_RDMLX_INBOUND_HTTP.TXT* into function **iiiFN09** following the comment * <<< Inbound binding logic goes here >>>

```
* Bind service to read HTTP request content
CHANGE FIELD(#JSMXCMD) TO('BIND
SERVICE(IIIPRO07_RESPONSE) TYPE(*INBOUND)
BINDTRACE(*YES)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
* Get fragment - EMPLOYEELIST
CHANGE FIELD(#JSMXCMD) TO('GET
FRAGMENT(EMPLOYEELIST) SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
* <<< Enter the fragment loop logic for EMPLOYEE >>>
BEGIN_LOOP /* EMPLOYEE */
* Get fragment - EMPLOYEE
CHANGE FIELD(#JSMXCMD) TO('GET FRAGMENT(EMPLOYEE)
SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
LEAVE IF(#JSMXSTS *EQ NOFRAGMENT')
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
* Get list - EMPLOYEESKILL
CHANGE FIELD(#JSMXCMD) TO('GET LIST(EMPLOYEESKILL)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
```

```
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG #EMPLOYEEESKILL)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
END_LOOP /* EMPLOYEE */
```

2. Change the working list name in the * Get list – EMPLOYEEESKILLS logic. The **TO_GET** keyword needs to refer to your working list **#EMPskill**. (Define at the top of the function the list as DEF_LIST NAME(#empskill) FIELDS(#skildesc) TYPE(*working) ENTRIES(9999)) Review the code just inserted and note that it includes all the response logic which was outlined in Step 5. i.e.
 - BIND to response service
 - GET employeelist fragment
 - .Loop to GET employee fragment, GET list employeeskill and build skills working list
3. You can improve this logic by handling the condition where an employee has no skills. Add a CONTINUE if JSMXSTS = NOLIST, following the * Get List – EMPLOYEEESKILL block of code. This block of code should now look like the following:

```
* Get list - EMPLOYEEESKILL
CHANGE FIELD(#JSMXCMD) TO('GET LIST(EMPLOYEEESKILL)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG #EMPLOYEEESKILL)
CONTINUE IF('#JSMXSTS *EQ NOLIST')
```
4. Your function currently loops and retrieves each employee fragment (EMPNO and FULLNAME) and a list of skills for that employee. Function iiiFN09 needs to save the fullname and skills data into a second working list, which will be returned to the calling display function iiiFN10. Add logic before the END_LOOP to read the #EMPskill list and add entries to working list #WL_EMPSKL. Your new code should look like the following:

```
SELECTLIST NAMED(#empskill)
ADD_ENTRY TO_LIST(#wl_empskl)
ENDSELECT
```
5. Add a RETURN command after the JSM CLOSE
6. The CHECK subroutine from the Studio generated code simply returns to the

menu if status is not OK. Replace the routine with the following logic, which will display messages containing the JSM status and the JSM message. This will be useful for testing purposes.

* Check routine

```
SUBROUTINE NAME(CHECK) PARMs((#JSMXSTS *RECEIVED)
(#JSMXMSG *RECEIVED))
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132)
*
IF COND('#JSMXSTS *NE OK')
#MSGDTA := 'Error Status Code: ' + #JSMXSTS
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
#MSGDTA := 'Error Message: ' + #JSMXMSG
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDL1)
TO_GET(#JSMXSTS #JSMXMSG)
ABORT
ENDIF
*
ENDROUTINE
```

Your completed RDMLX code might appear as follows:

```
FUNCTION OPTIONS(*DIRECT) RCV_LIST(#wl_empskl)
* The following fields are used by the xml binding map
* #DEPARTMENT
* The following fragments are used by the xml binding map
GROUP_BY NAME(#DEPTREQ) FIELDS(#DEPARTMENT)
DEF_LIST NAME(#empskill) FIELDS(#skildesc) TYPE(*working)
ENTRYS(9999)
DEF_LIST NAME(#WL_EMPSKL) FIELDS(#FULLNAME #SKILDESC)
TYPE(*WORKING) ENTRYS(9999)
* Open service
USE BUILTIN(JSMX_OPEN) TO_GET(#JSMXSTS #JSMXMSG
#JSMXHDL1)
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMXSTS
#JSMXMSG)
* Load service
CHANGE FIELD(#JSMXCMD) TO('SERVICE_LOAD
SERVICE(HTTPOutboundXMLBindService) TRACE(*YES)')
```

```
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Bind service to create HTTP request content

```
CHANGE FIELD(#JSMXCMD) TO('BIND
SERVICE(IIIPRO07_REQUEST) TYPE(*OUTBOUND)')
```

```
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Set fragment - DEPARTMENTREQUEST

```
CHANGE FIELD(#JSMXCMD) TO('SET
FRAGMENT(DEPARTMENTREQUEST)
SERVICE_EXCHANGE(*FIELD)')
```

```
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Write content

```
CHANGE FIELD(#JSMXCMD) TO('WRITE INDENT(*YES)
BINDTRACE(*YES)')
```

```
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* Send HTTP request content

```
CHANGE FIELD(#JSMXCMD) TO('SEND HOST(10.44.10.236:80)
URI(/cgi-bin/jsmdirect?IIIFN08_SERVICE)')
```

```
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)
```

* <<< Inbound binding logic goes here >>>

* Bind service to read HTTP request content

```
CHANGE FIELD(#JSMXCMD) TO('BIND
SERVICE(IIIPRO07_RESPONSE) TYPE(*INBOUND)
BINDTRACE(*YES)')
```

```

USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)
* Get fragment - EMPLOYEEELIST
CHANGE FIELD(#JSMXCMD) TO('GET FRAGMENT(EMPLOYEEELIST)
SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)
* <<< Enter the fragment loop logic for EMPLOYEE >>>
BEGIN_LOOP /* EMPLOYEE */
* Get fragment - EMPLOYEE
CHANGE FIELD(#JSMXCMD) TO('GET FRAGMENT(EMPLOYEE)
SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
LEAVE IF('#JSMXSTS *EQ NOFRAGMENT')
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)
* Get list - EMPLOYEEESKILL
CHANGE FIELD(#JSMXCMD) TO('GET LIST(EMPLOYEEESKILL)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG #EMPSKILL)
CONTINUE IF('#JSMXSTS *EQ NOLIST')
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)
SELECTLIST NAMED(#empskill)
ADD_ENTRY TO_LIST(#wl_emp skl)
ENDSELECT
END_LOOP /* EMPLOYEE */
* Close binding
CHANGE FIELD(#JSMXCMD) TO('CLOSE')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1
#JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMXSTS
#JSMXMSG)* Unload service
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1

```

```

'SERVICE_UNLOAD') TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMXSTS
#JSMXMSG)
* Close service
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDL1)
TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMXSTS
#JSMXMSG)
RETURN
* Check routine
SUBROUTINE NAME(CHECK) PARMs((#JSMXSTS *RECEIVED)
(#JSMXMSG *RECEIVED))
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132)
*
IF COND(#JSMXSTS *NE OK')
*
#MSGDTA := 'Error Status Code: ' + #JSMXSTS
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
#MSGDTA := 'Error Message: ' + #JSMXMSG
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDL1)
TO_GET(#JSMXSTS #JSMXMSG)ABORT
*
ENDIF
*
ENDROUTINE

```

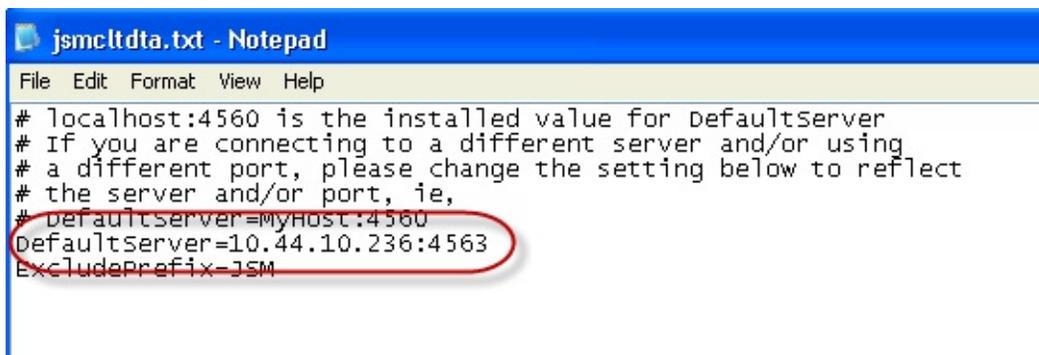
6. Save and compile the function and check in to the server.

Step 7. Test iiiFN08 Server with Client functions

In this step, you will use **iiiFN10** (your user interface) to invoke **iiiFN09** (your client function), which will in turn send an XML request to **iiiFN08** (the server function) via the service you created in INT008. The client function **iiiFN10** will then process the server's response and display it to the user.

If you are using a JSM Server on an IBM i, then you have two ways to test your client functions:

- Check the functions into the IBM i and run your test on the IBM i
or
- Set up the **jsmcltdta.txt** file to point to the JSM Server on the IBM i. This file is located in your Visual LANSA folder, by default: C:\Program Files\LANSA\X_WIN95\X_LANSA. See example:



```
jsmcltdta.txt - Notepad
File Edit Format View Help
# localhost:4560 is the installed value for Defaultserver
# If you are connecting to a different server and/or using
# a different port, please change the setting below to reflect
# the server and/or port, ie,
# defaultserver=myhost:4560
Defaultserver=10.44.10.236:4563
ExcludePrefix=JSM
```

You can then run your client functions locally in Windows and iiiFN09 will be using the JSM Server on the IBM i. This arrangement is very convenient for testing and using debug.

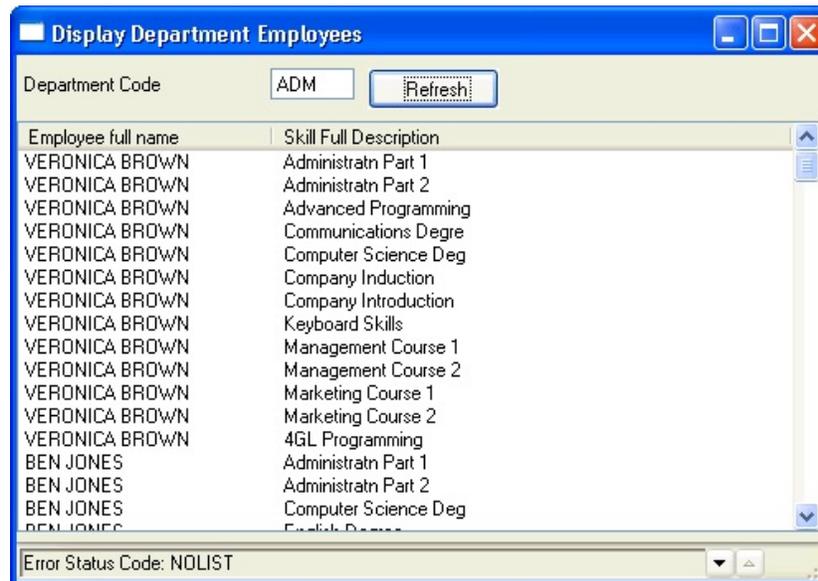
1. Running as a Windows application, you can either execute the process iiiPRO07 or the display function iiiFN10. In either case the process runs first as a graphical menu



2. Double click on *Display Department & Employee Client* icon to run function iiiFN10.
3. Enter a department code (e.g. ADM or FLT, etc) and hit Enter. A NOLIST message may be shown if an employee with no skill records is found.
4. Use the Forward and Bkwd buttons to browse through the list of employees. You could compare the results with the User Agent test in INT008.

Step 8. A Client Visual Form (optional)

If a Windows form is required to serve as the client application, this can easily be written. It would also call the client RDMLX function iiiFN09 to handle the interaction with the server function iiiFN08.



1. Create a new basic form - **iiiFRM01 – Display Department Employees**.

Copy the following code to replace the existing form code:

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(319)
Clientwidth(492) Height(353) Left(337) Top(148)
Define_Com Class(#DEPARTMENT.Visual) Name(#DEPARTMENT)
Displayposition(1) Height(19) Left(4) Parent(#COM_OWNER)
Tabposition(1) Top(8) Usepicklist(False) Width(201)
Define_Com Class(#PRIM_LTVW) Name(#LTVW_1) Componentversion(2)
Displayposition(2) Fullrowselect(True) Height(249)
Keyboardpositioning(SortColumn) Left(0) Parent(#COM_OWNER)
Showsortarrow(True) Tabposition(2) Top(40) Width(489)
Define_Com Class(#PRIM_LVCL) Name(#LVCL_1) Displayposition(1)
Parent(#LTVW_1) Source(#FULLNAME) Width(33)
Define_Com Class(#PRIM_LVCL) Name(#LVCL_2) Displayposition(2)
Parent(#LTVW_1) Source(#SKILDESC) Width(44) Widthtype(Remainder)
Define_Com Class(#PRIM_PHBN) Name(#PHBN_1) Caption('Refresh')
Displayposition(3) Left(213) Parent(#COM_OWNER) Tabposition(3) Top(8)
```

```
Define_Com Class(#PRIM_STBR) Name(#STBR_1) Displayposition(4)
Height(24) Left(0) Messageposition(1) Parent(#COM_OWNER)
Tabposition(4) Tabstop(False) Top(295) Width(492)
Def_List Name(#WL_EMPSKL) Fields(#FULLNAME #SKILDESC)
Type(*WORKING) Entrys(9999)
Evroutine Handling(#com_owner.Initialize)
Set Com(#com_owner) Caption(*component_desc)
Endroutine
```

```
Evroutine Handling(#PHBN_1.Click)
Clr_List Named(#wl_empskl)
Exchange Fields(#deptment)
Call Process(*direct) Function(iiiFN09) Pass_Lst(#WL_EMPSKL)
Clr_List Named(#ltvw_1)
Selectlist Named(#wl_empskl)
Add_Entry To_List(#ltvw_1)
Endselect
Endroutine
End_Com
```

2. Change the CALL statement to call your function iiiFN09.
3. Compile and run the form. You should either be using a local web server and JSM Server or have the file: jsmcltdta.txt configured to point to the JSM Server on an IBM i.

Summary

Important Observations

- The HTTPInboundXMLBindService and HTTPOutboundXMLBindService can be used effectively on opposite sides of a transaction.
- The HTTPOutboundXMLService is useful when *POSTing content.
- The same XML must be used to create both sides of the client/server transaction.
- Wherever one side creates inbound bindings, the other side creates outbound bindings (and vice versa). The same complementary rule is true for GET and SET commands.

Tips & Techniques

- Use RDML wrapper functions to invoke more complex RDMLX functions
- It is useful to code each side of a client/server interaction separately, but keeping in mind exactly what is expected from each side.

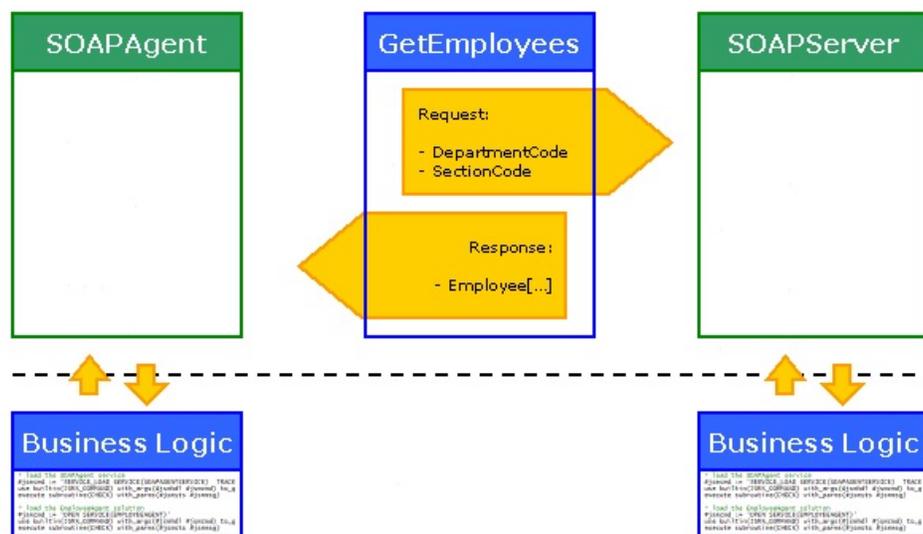
What I Should Know

- How to create a client-side RDMLX function using HTTPOutboundXMLBindService
- How to code and understand both sides of an XML interaction.
- How to test a server-side function using a complementary client-side function.

INT010A - SOAP Service - Define Server

There are three tutorials in this set which together, demonstrate how to implement a web service using LANSAs Integrator Studio. These tutorials need to be created in sequence.

These tutorials create a web service called **GetEmployees**.



- The SOAP Agent, or client, consuming the **GetEmployees** web service will pass a department and section to the server.
- A list of employees for that department and section will be returned by the publisher, the SOAP Server, as an array.
 - If only a department is passed, a list of employees in all sections for that department will be returned.
 - If no department and section is provided then all employees for the company will be returned.

To create the service as shown in the diagram above, you must complete three tutorials:

- [INT010A - SOAP Service - Define Server](#)
- [INT010B - SOAP Service - Define Agent](#)
- [INT010C - SOAP Service - Test](#)

The Business Logic (program code), provided for you in **INT010C** to test this service has been created as a Visual LANSAs form. In LANSAs it could also be written as a 5250 function, or as a web application (WAM). You could also use a

3GL such as ILE RPG.

The following summarizes the steps you will take in the three tutorials to build and test your SOAP Service:

INT010A – SOAP Service – Define Server

- Define SOAP Server Types – the variables to be handled by this service
- Create a SOAP Server solution using the *SOAP Server Wizard*.
- Map parameters to program variables
- Build the solution, creates WSDL and jar file for `iiiEmployeeServer`.
- The WSDL defines the `GetEmployees` SOAP service
- Use the generated RDMLX to create function which supports the `iiiEmployeeServer` service and add your business logic
- Register the function as a JSMDIRECT service

INT010B – SOAP Service – Define Agent

- Create a new solution using the *SOAP Agent Wizard*
- Define the SOAP Agent `iiiEmployeeAgent` based on the WSDL created in INT010A
- Map parameters to program variables
- Build the solution creates a jar file, properties file and sample RDMLX
- Create the form `iiiFRM02` based on the generated RDMLX and add your business logic

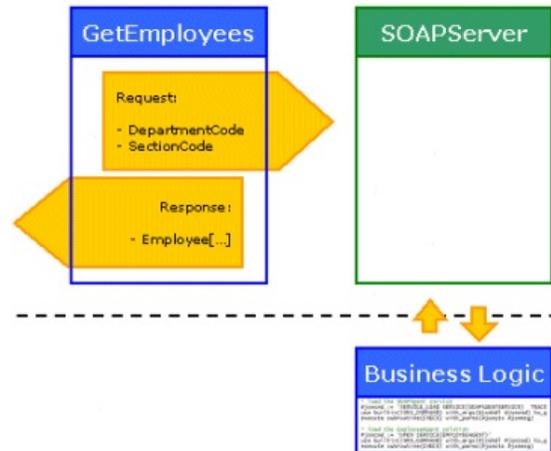
INT010C – SOAP Service – Test

- The form accepts department and section code and runs the `IIIEMPLOYEEAGENT` service on the server
- Sets `IIIEMPLOYEEAGENT` operation to `GETEMPLOYEES`
- Sets parameters to program variables
- CALLs the service
- Web Server invokes JSMDIRECT which runs the function implementing `IIIEMPLOYEESEVER`
- Form receives employees list and populates list view to display results.

Objectives:

- To create a SOAP Server Project that publishes a web service that other applications can use.

- To define the SOAP Server Solution using LANSa Integrator Studio.
- To create the Server side of a web service called **GetEmployees**.



To achieve these objectives, you will complete the following:

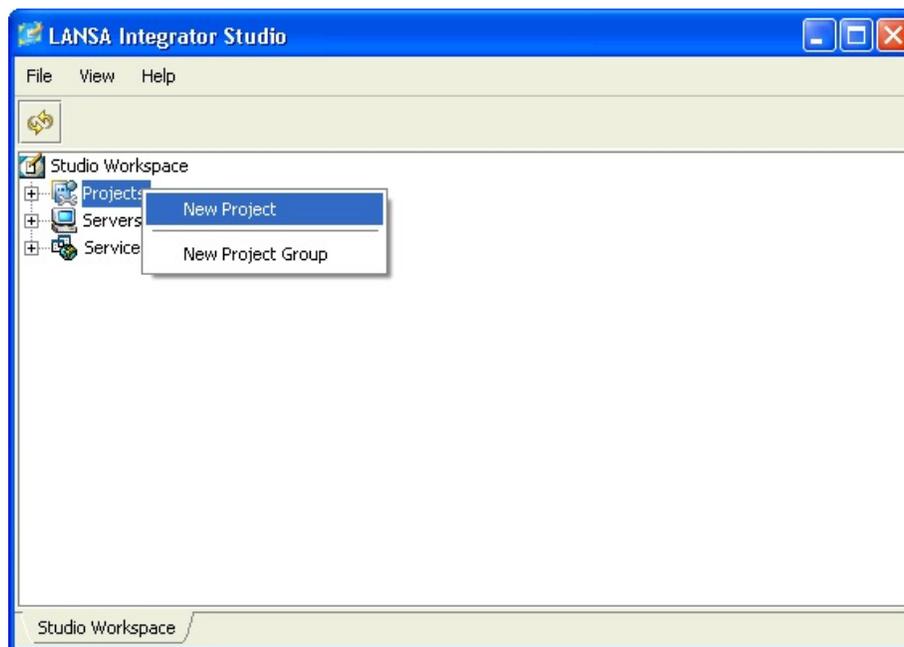
- [Step 1. Create a New project](#)
- [Step 2. Define custom SOAP Server Types](#)
- [Step 3. Create a new SOAP Server Solution](#)
- [Step 4. Create a new SOAP Operation](#)
- [Step 5. Map Parameters to Program Variables](#)
- [Step 6. Build the SOAP Server Solution](#)
- [Step 7. Deploy the SOAP Server Solution](#)
- [Step 8. Create the SOAP Server Business Logic](#)
- [Summary](#)

Step 1. Create a New project

In this step you will use the *LANSA Integrator Studio* to create a new Project to contain your SOAP Server Solution.

A Project is simply a way to organize your work. A Project contains a collection of files related to a particular application or task using LANSa Integrator. Within a Project will be one or more Solutions.

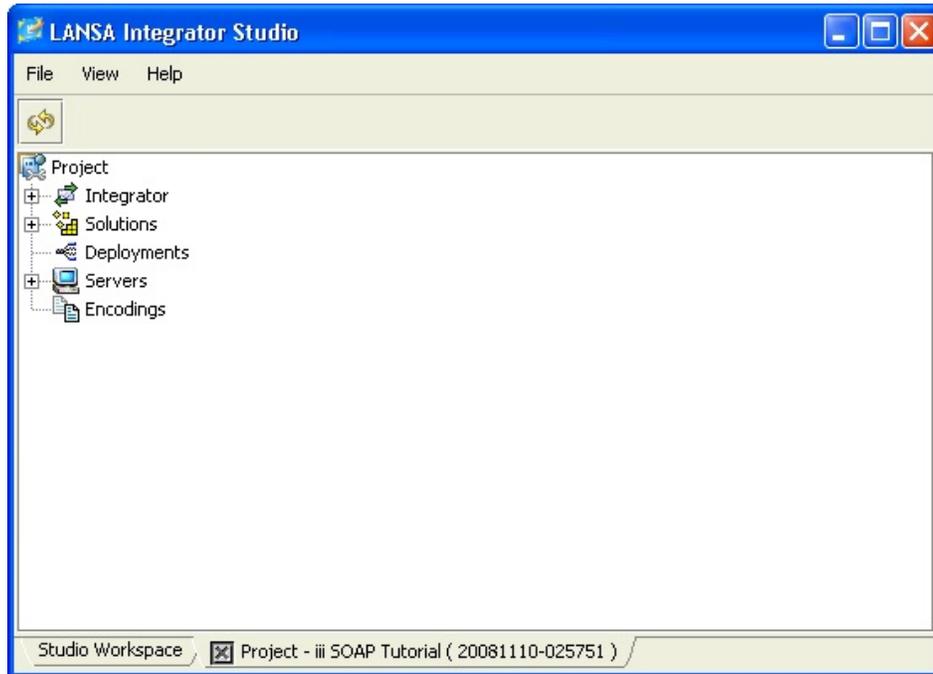
1. Start LANSa Integrator Studio from the desktop icon.
2. Create a *New Project*:
 - a. Display the Studio Workspace tab.
 - b. Select and right click on *Projects* to open the context menu.



- c. Select *New Project* to open the *LANSA Integrator Studio* dialog box.
3. Enter the name of your new project, in this case **iii SOAP Tutorial where iii are your unique three letters**, in the *LANSA Integrator Studio* dialog box and press *OK*.



A new tab will appear at the bottom of the studio workspace for the project.



Notice that *Integrator Studio* has assigned a unique identifier to the project based on the date and time you created the project. This identifier is important later when it comes to retrieving, modifying and deploying the properties for the solutions contained in the project.

Step 2. Define custom SOAP Server Types

SOAP Server Types are user-defined data types used in a SOAP server to define request and response parameters. They can consist of one or more elements, each of which may be of a simple type such as strings or numbers.

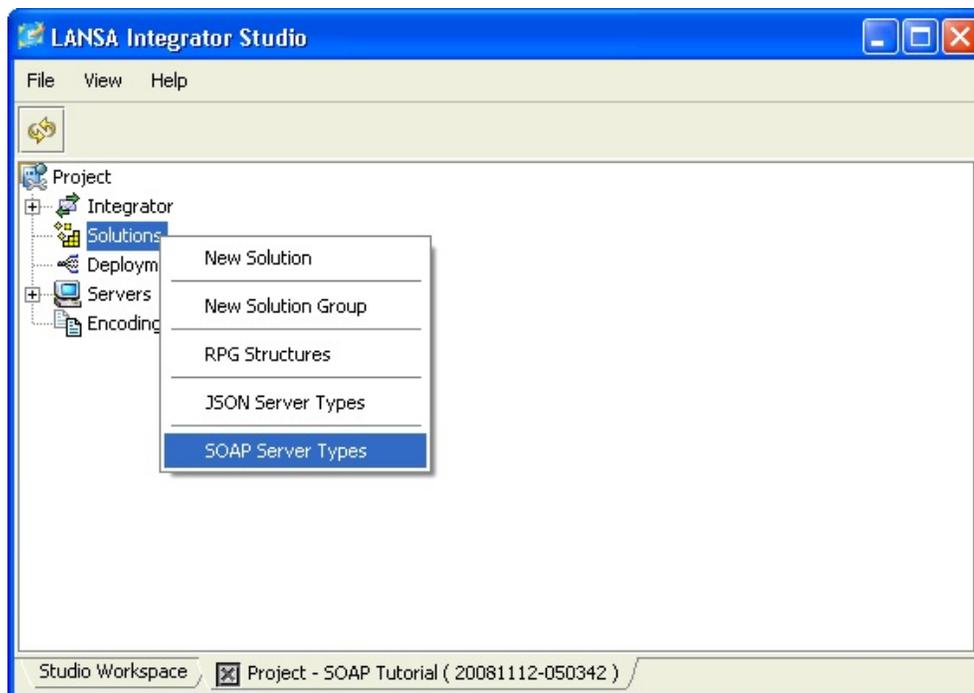
Defining them using LANSА Integrator Studio makes them independent of the web services that uses them.

In this step you will define the Solution Employee server type for **GetEmployees**.

GetEmployees will return these four elements:

- employee id
- first name
- surname
- salary.

1. Select the tab for your *Project* in the LANSА Integrator Studio.
2. Select *Solutions* and right mouse click to open the context menu.



3. Select the *SOAP Server types* option from the menu.
This will open the *LANSА Integrator SOAP Type Editor*.

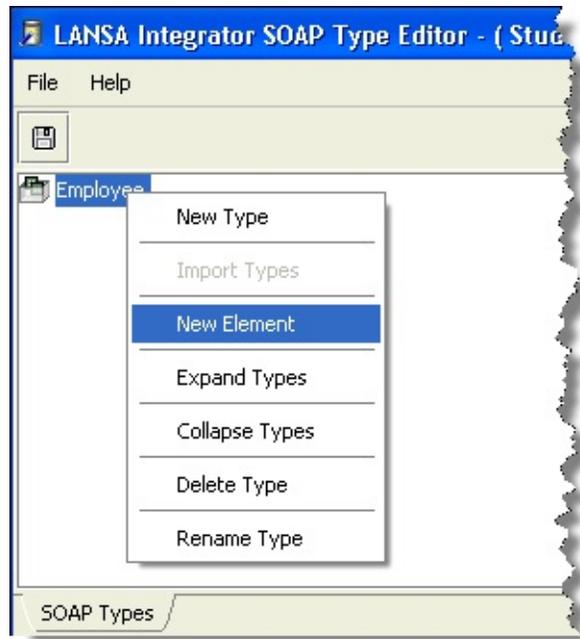
4. Right click in the empty space to open the context menu and then select *New Type*.



The *LANSA Integrator SOAP Type Editor* dialog is displayed.



5. Enter the name for the new type in the *LANSA Integrator SOAP Type Editor* dialog box and click *OK* (or press the *Enter* key).
6. Now you must define the elements that make up the **Employee** Type.



To define an element:

- a. Right click on **Employee** to open the context menu.
- b. Select *New Element* from the context menu.

The *Element* dialog box is opened.

7. Enter **employeeId** as the Element Name of the first *Element*. Note the case of this entry. The first character is lower case. The first character of other words (in this case **Id**) is upper case.

Select the *Value String* from the drop down list.

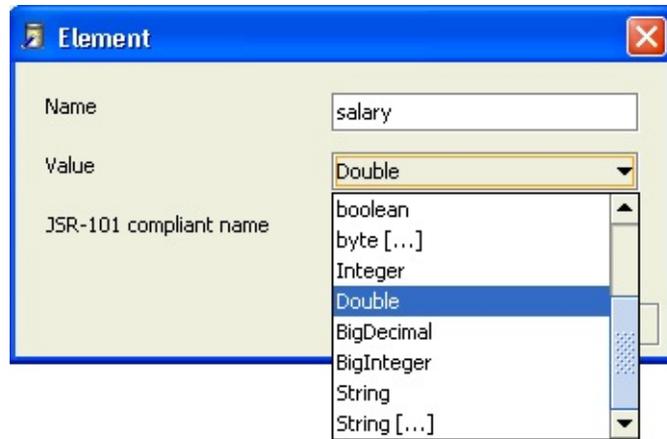
8. Press *OK*.

The dialog remains open ready for you to enter any further elements.

9. Enter the second element in the same way as for employeeId:

That is, **firstName** with a *Value* of **String**.

10. Create the third element, the **salary** element.



The *Value* for salary is the appropriate numeric type which is **double**.

11. Create the final element, **surname** with a *Value* of **String**.

12. When you have finished adding elements, close the *Element* dialog box by clicking the *Cancel* button.

You have created the *SOAP Server Types* necessary to support your web service.

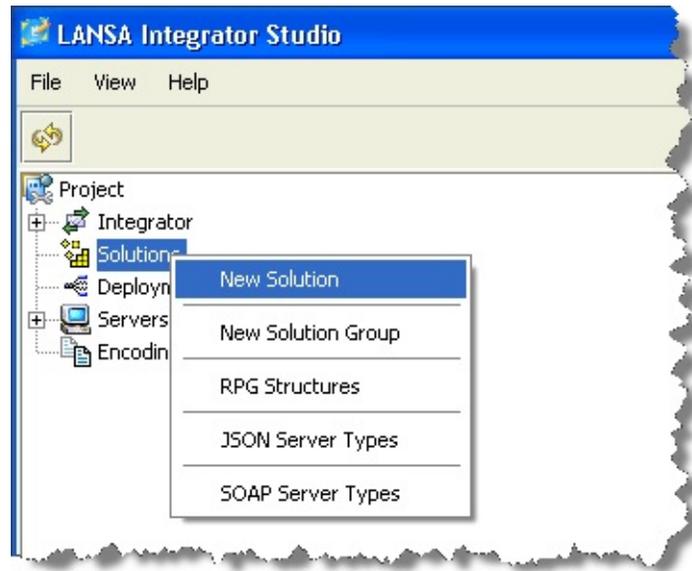
13. Save the changes using either the  *Save* button, the *Ctrl + S* keys or *Save* from the *File* menu.



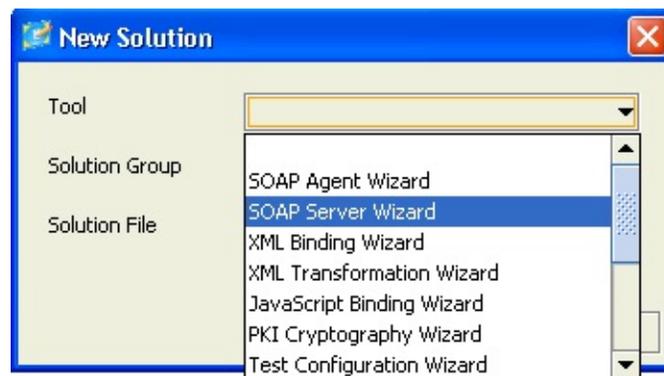
14. Close the *LANSA Integrator SOAP Type Editor* using *Exit* from the *File* menu or the *Alt + F4* keys.

Step 3. Create a new SOAP Server Solution

1. With your **iii SOAP Tutorial** project open in Studio.
2. Select the *Solutions* folder and right click on it to open the context menu.



3. Select *New Solution* from the context menu to open the *New Solution* dialog box.



4. In the *New Solution* dialog box, select the **SOAP Server Wizard** Tool from the drop-down list.



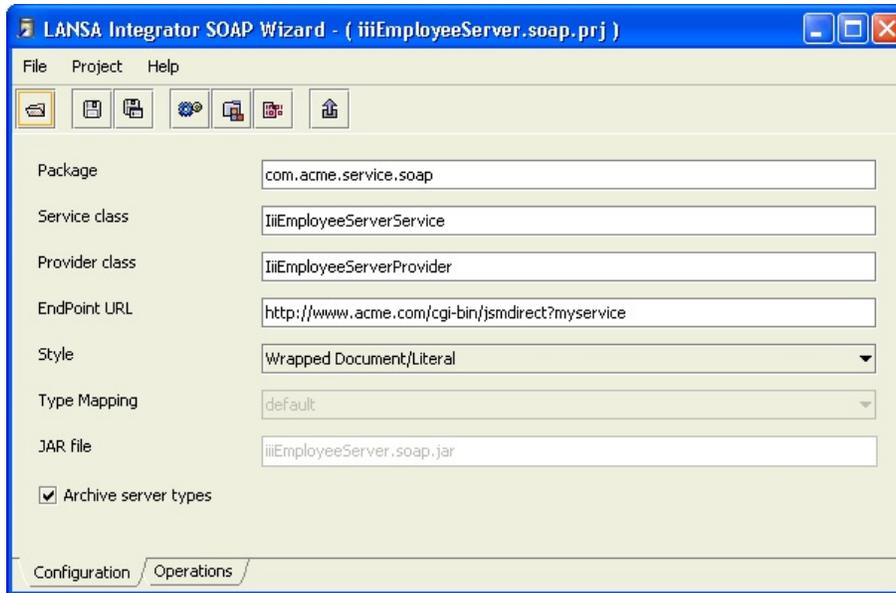
5. Enter the name of the *Group* to contain this Solution, in this case **EmployeeServer**.

If you had more than one group already, you could select an existing *Solution Group* from the drop-down list.

6. In *File*, enter a name for the *Solution*, in this case **iiiEmployeeServer**. This name will be used for the project (prj) file. It will also be used by the SOAP Wizard as a prefix for some of the objects that it generates for the solution.
7. Press *OK*.



8. Select *Yes*, to confirm that the new solution group is to be created.
LANSa Integrator opens the *SOAP Wizard*.
Select the *Configuration* tab.



LANSa Integrator provides default values but you must change some of them to be specific to this project and the server that will service it.

9. Change these values:

a. **Package name**

This is the name that the SOAP Wizard will use when generating the Java implementation files for your solution so make it something unique to your solution. For this tutorial, enter **com.iiicompany.service.soap**

b. **EndPoint URL**

This identifies the server to which users of this web service need to send their web service requests and it will be written into the WSDL file that describes this service.

For this tutorial, if you are using an IBM i JSM Server, then the server will be on the IBM i and the client will be on your PC. In this case the URL will point to the IBM i server's IP address. For example:
`http://10.44.10.236:80/cgi-bin/jsmdirect?iiiemployeeserver`

If you are using a local Windows web server and JSM Server, your URL may look like this:

`http://localhost/cgi-bin/jsmdirect.exe?iiiemployeeserver`

In this case your client and server will both be running locally.

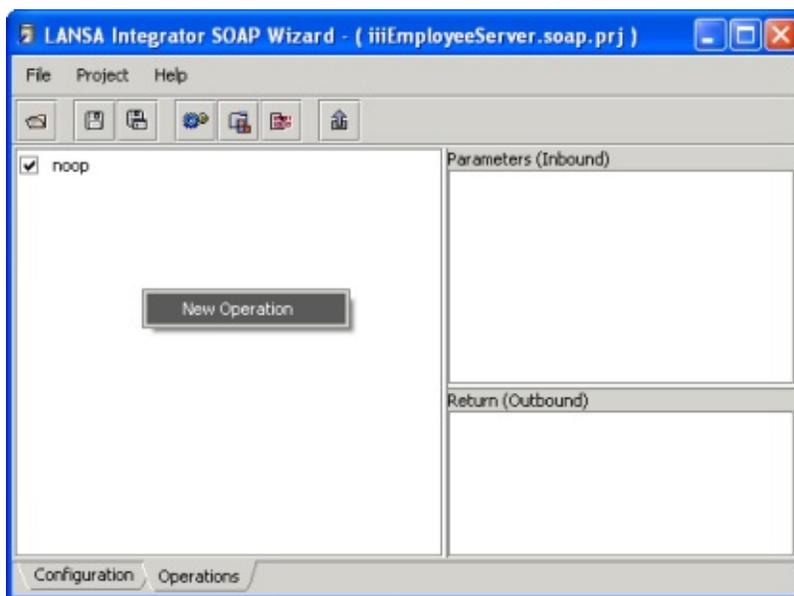
c. Save your changes. Do not close the *SOAP Wizard*.

Step 4. Create a new SOAP Operation

In this step you will create the *SOAP Operation* for your web service, **GetEmployees**, and define its *Inbound* and *Outbound* parameters.

Remember that **GetEmployees** will receive parameters that specify a department code and section code and will return a list or array of details for all the employees in that department and section.

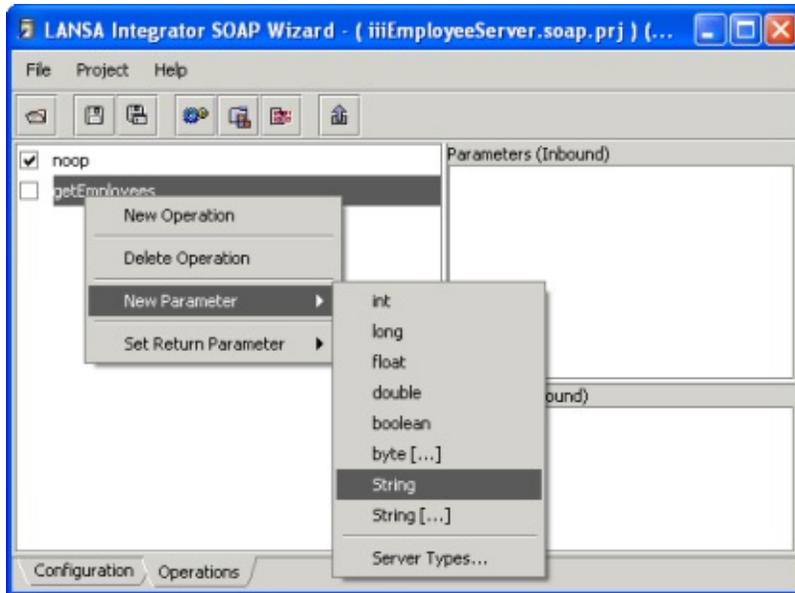
1. Select the *Operations* tab of the *SOAP Wizard*.
2. Right click in the *Operations* panel to open the context menu.



3. Select *New Operation* from the context menu.



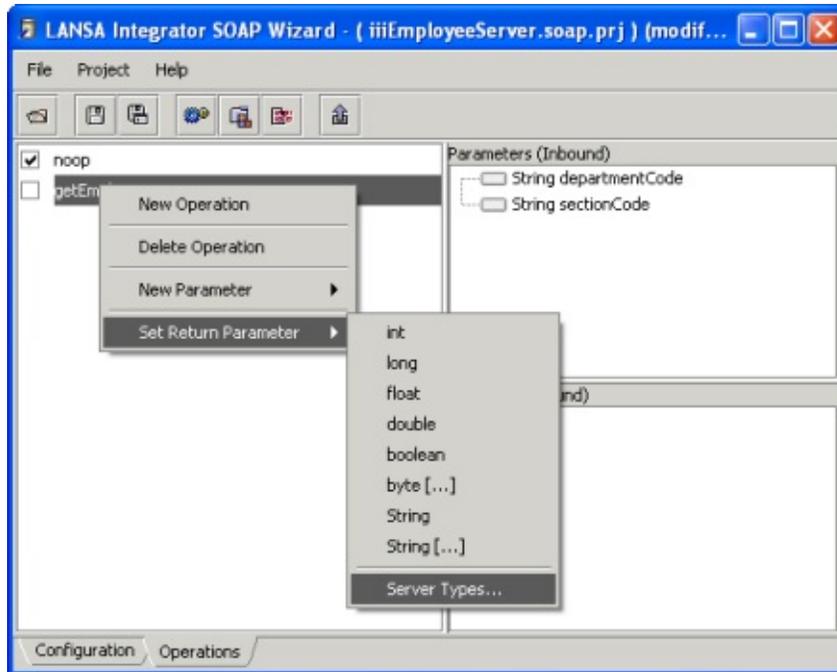
4. In the resulting dialog, enter the *Operation Name*, **getEmployees** and click *OK*.
5. Now you will define the Inbound parameters. To do so:
 - a. Right click on the **getEmployees** Operation to open the context menu.
 - b. Select *New Parameter* from the context menu.



- c. Select the type for the parameter, in this case, a **String**.
6. In the resulting dialog, enter the *Parameter name*, in this case **departmentCode** and click *OK*.



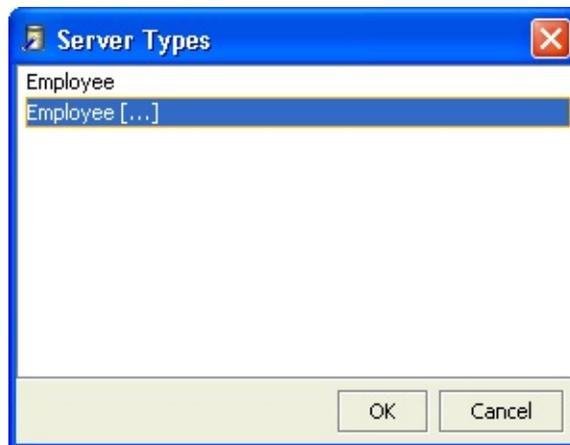
7. Now repeat the same sequence for the remaining inbound parameter **sectionCode**, which is also a **String** and click *OK*.
8. Define the **return** or *Outbound* parameter. This follows a similar procedure to the *Inbound* parameters.
 - a. Right click on **GetEmployees** to open the context menu.



- b. Select *Set Return Parameter* from the context menu and select *Server Types...* from the sub menu.

Server Types is selected because your return parameter is going to be the Server Type you created in [Step 2. Define custom SOAP Server Types](#).

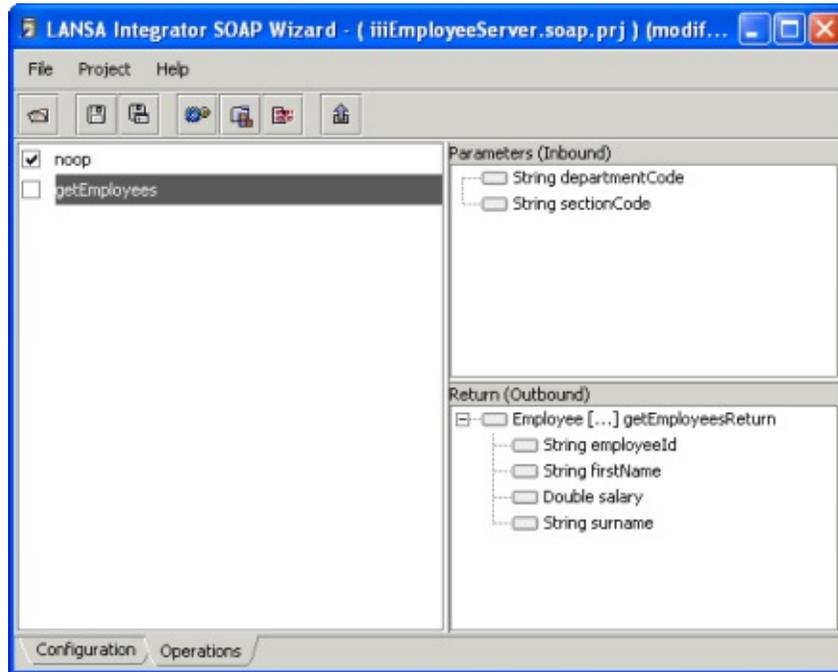
9. In the *Server Types* dialog box, there are two entries for your Employee Server Type. The first one represents a single instance of Employee. The second one with [...] represents an array of Employees.



Because you are returning an array of employees, select **Employee [...]** and click *OK*.

You are returned to *Operations* tab of the *SOAP Wizard* where you can see the inbound and outbound parameters that you have defined. If you expand

the entries in the Employee return parameter you will see the elements that make up the Employee server type.



You have now defined your SOAP Operation and its parameters.

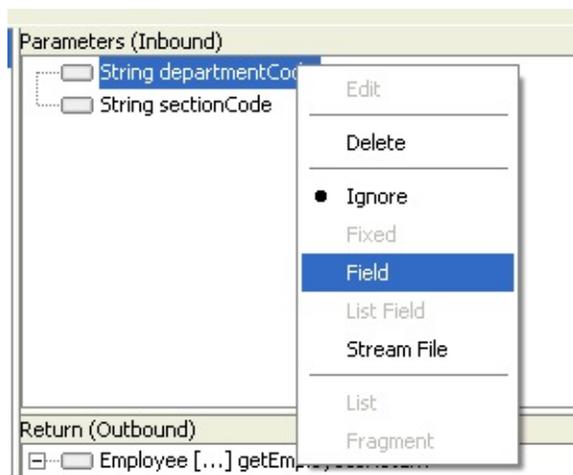
When you define your own web services, you will probably define more than one SOAP Operation, but you will only create one for this tutorial.

10. Save your changes.

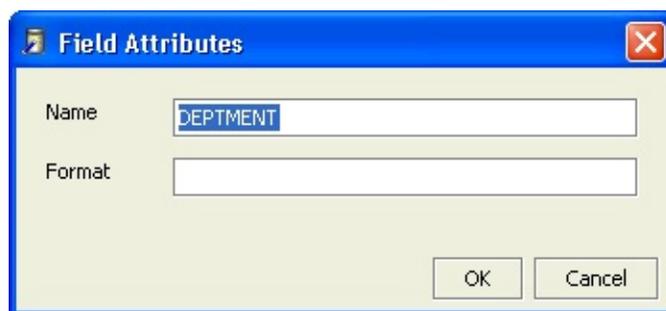
Step 5. Map Parameters to Program Variables

In this step, you will map the web service *Inbound* and *Outbound Parameters* you created in [Step 4. Create a new SOAP Operation](#), to the program variable names that you will use in the program that implements this web service.

1. Map the *Inbound* parameters:
 - a. Select and right click on the `departmentCode` to open the context menu.

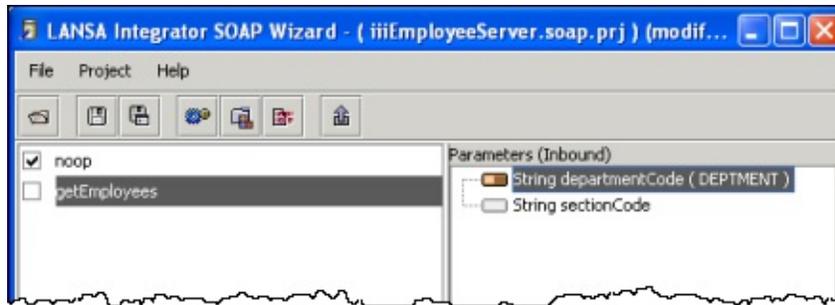


- b. Select *Field* from the context menu to open the *Field Attributes* dialog box.



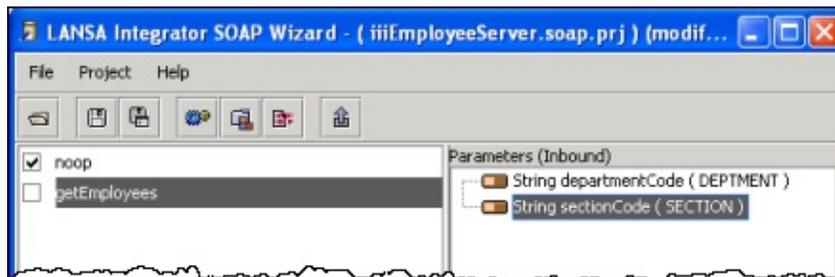
- c. In the *Field Attributes* dialog box enter the corresponding program variable name, in this case **DEPTMENT**, and click *OK*. Leave *Format* blank.

The mapped program variable names you entered are shown in brackets after the parameter name. Note that the icon has changed to indicate that this parameter is mapped.



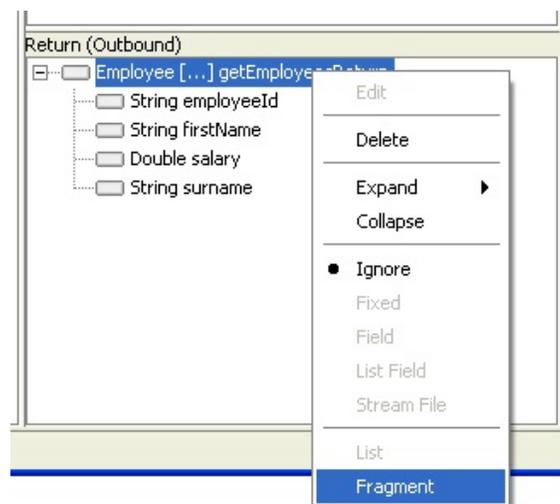
2. Repeat this procedure for the second *Inbound* parameter, sectionCode. Select *Field* from the context menu and map to the program variable name **SECTION**.

Note how the variable name is shown with the parameter name and that the icon has changed.



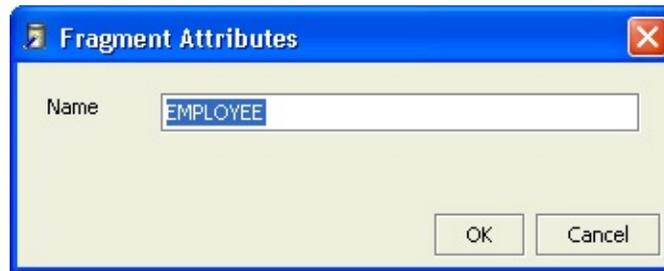
3. Map the *Return* or *Outbound* parameter. Because the return parameter is a complex type, the procedure is slightly different:

- a. Select and right click the **Employee** parameter to open the context menu.

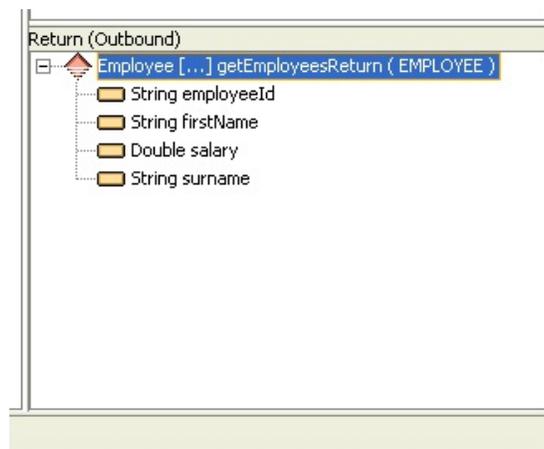


- b. Select *Fragment* from the context menu. This option allows you to receive all the elements of Employee in a single call.

- c. In the resulting dialog, *Fragment Attributes*, enter the name you wish to give to the Fragment, in this case, **EMPLOYEE**.

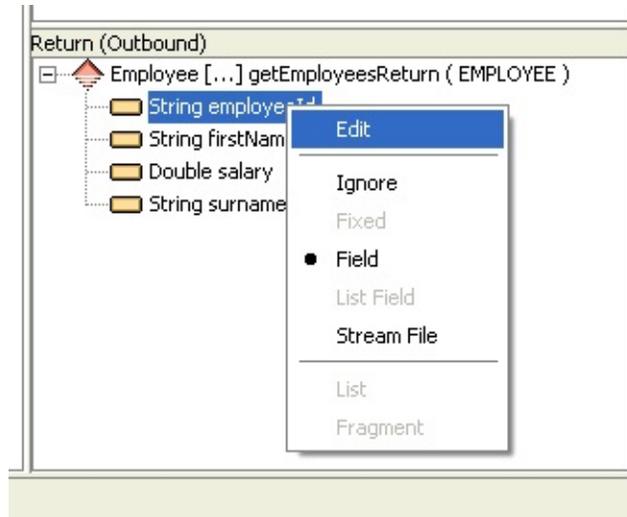


- d. Click *OK* to confirm this assignment.

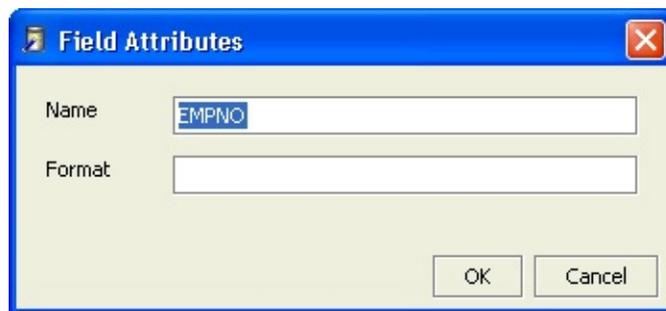


The mapped fragment name is now shown beside the Employee return parameter, and the icon has changed. In this case, the icon has changed for all the elements of employee because when you marked Employee as a fragment, LANSa Integrator assumed that all the elements would be mapped to fields. All that remains to be done is to assign the program variable names to the elements within Employee.

4. Complete the mapping for the outbound elements in Employee:
- Select and right click the **employeeId** element to open the context menu.
 - Select *Edit* from the context menu.



- c. Enter the program variable name, **EMPNO**, in the *Field Attributes* dialog box and click *OK*.



- d. Now you can repeat the procedure for the remaining elements:

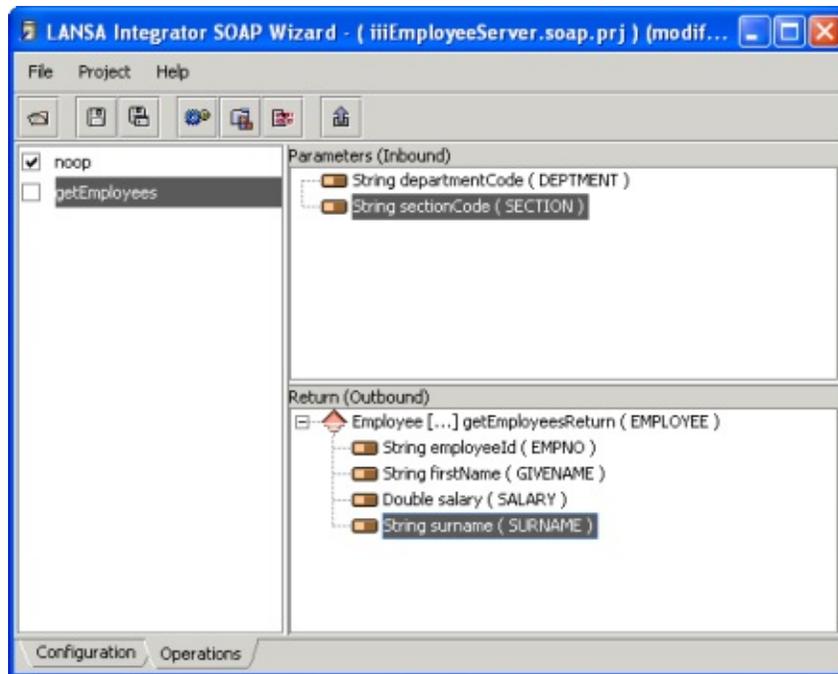
element program variable

firstName GIVENAME

salary SALARY

surname SURNAME

The mapping is now complete. Your window should look like the one in the screen capture following.

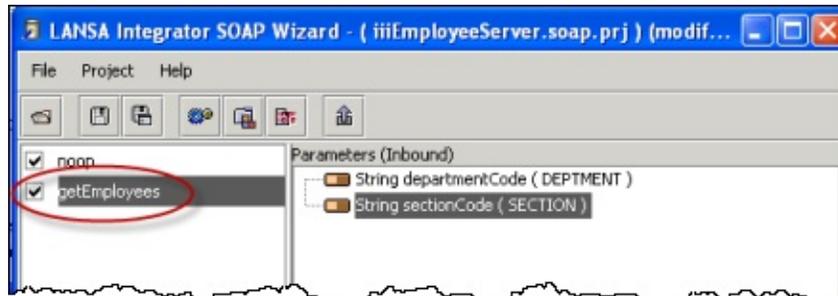


5. Save your changes.

Step 6. Build the SOAP Server Solution

In this step you will build the *SOAP Server* solution and then review the files that LANSAs Integrator has generated.

1. Build the solution:
 - a. Check (i.e. P) the box beside the Operations to be included in the build. In this case, you will tick the box next to the **getEmployees** Operation.

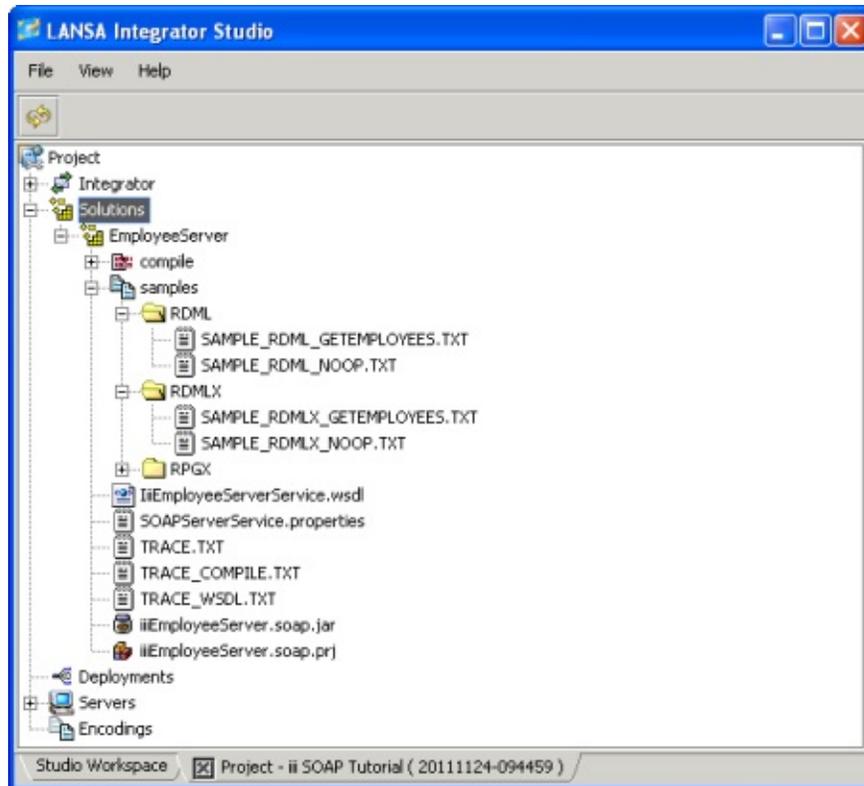


- b. Select the  *Build* button on the toolbar to start the build.
When the build has been completed, this message is displayed.



- c. Click *OK* to continue.
2. Save  your new Solution and *OK* at the confirm dialog.
3. Exit the *SOAP Wizard*.

You are returned to the *iii SOAP Tutorial Project* tab in the LANSAs Integrator Studio.
4. Expand the Project's *Solutions* folder so that you can see the files that LANSAs Integrator has generated.



The files that have been generated are:

- A **jar** file, **iiEmployeeServer.soap.jar**
This file contains the compiled classes, ready for you to deploy to the server.
- A project (**prj**) file, **iiEmployeeServer.soap.prj**.
If you need to use the SOAP Wizard again to revise what you have done, you will open it using this prj file.
- The Web Service Definition Language file (**wsdl**), **iiEmployeeService.wsdl**
This file describes this web service in a common XML format which enables **any** application to use this web service. You will need the file name and path of this file in [Step 7. Deploy the SOAP Server Solution](#).
- A **properties** file, **SOAPServerService.properties**
You will deploy this file to the web server in order to fully enable this web service.
- Other files consisting of some sample generated RDML and RDMLX code for an application that implements this web service and trace and log files.

Step 7. Deploy the SOAP Server Solution

You need a server in order to complete this step. If a server is not available, you can complete this step, except for Publishing the files.

In the preceding step, you built the files for the SOAP Server Solution in your development environment. They now need to be moved, or deployed, to the JSM Server.

For this tutorial, in [Part A. Publish Files](#) you will:

- Publish the **SOAPServerService.properties** file to associate your solution with the generated java classes that implement it.
- Publish the Java classes that are in the **iiiEmployeeServer.SOAP.jar** file.

In [Part B. Define Service entry](#) you will:

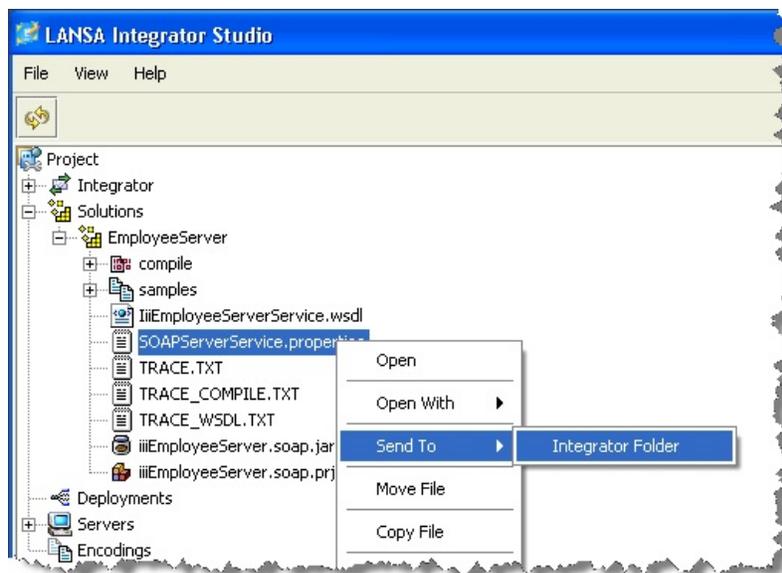
- Create the JSMDirect Service entry that will associate the web service name with the business logic that implements it.

Note: If a service is to be accessed by external parties, you would also need to deploy the WSDL file that describes the service.

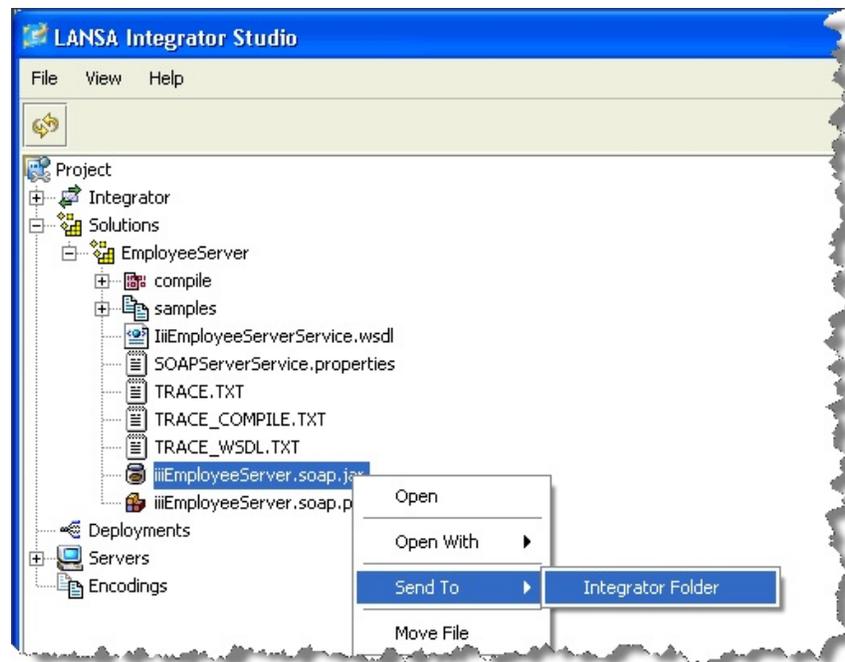
Part A. Publish Files

In these next steps, you will move the properties file to Integrator and then to the JSM server.

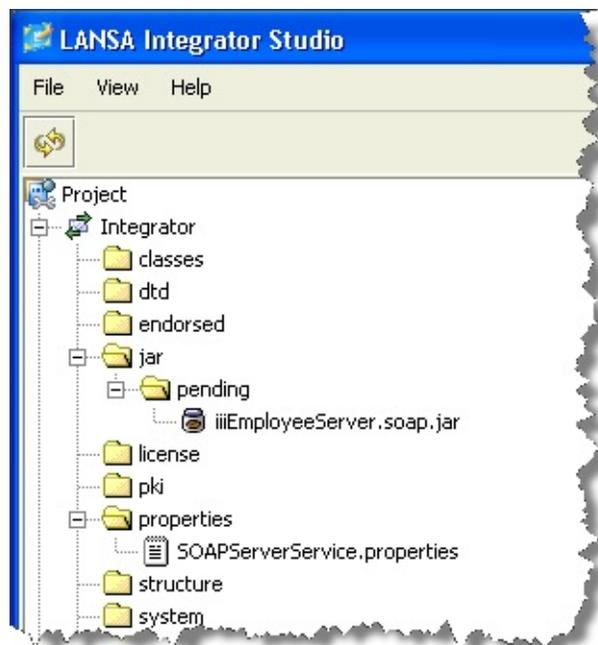
1. Display the Project tab for your project in the *LANSA Integrator Studio*.
So that you can see your project's files, expand the *Solutions* folder, then expand the *EmployeeServer* folder.
2. Copy the **properties** file to the Integrator project folder. To do this:
 - a. Highlight and right click the **SOAPServerService.properties** file to open the context menu.



- b. Select *Send To* and then *IntegratorFolder* from the context menu.
3. Copy the **jar** file to the Integrator project folder. To do this:
 - a. Highlight the **IiiEmployeeServer.SOAP.jar** file to open the context menu.

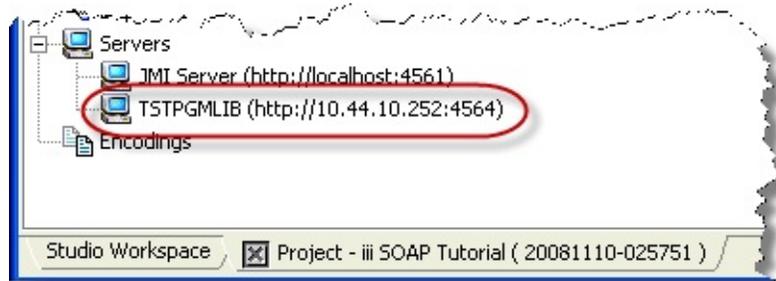


- b. Select *Send To* and then *Integrator Folder* from the context menu.
4. Contract the *Solutions* folder and expand the *Integrator* folder.
 Note the location of your *Solution* files in the *Integrator* project folder

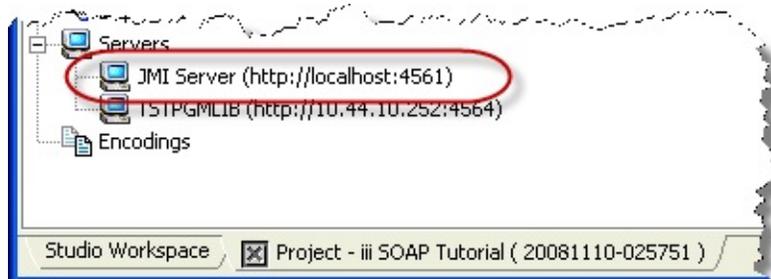


5. From your project tab in Studio, open your server instance so that you can move the properties and jar files to it.
 - a. Your server will already be defined from earlier exercises in this course.

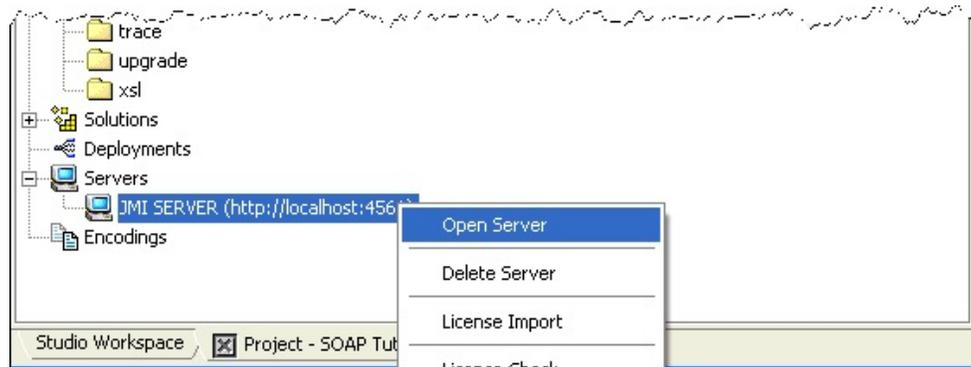
Your course may be using an IBM i JSM Server, in which case the definition will look similar to the following:



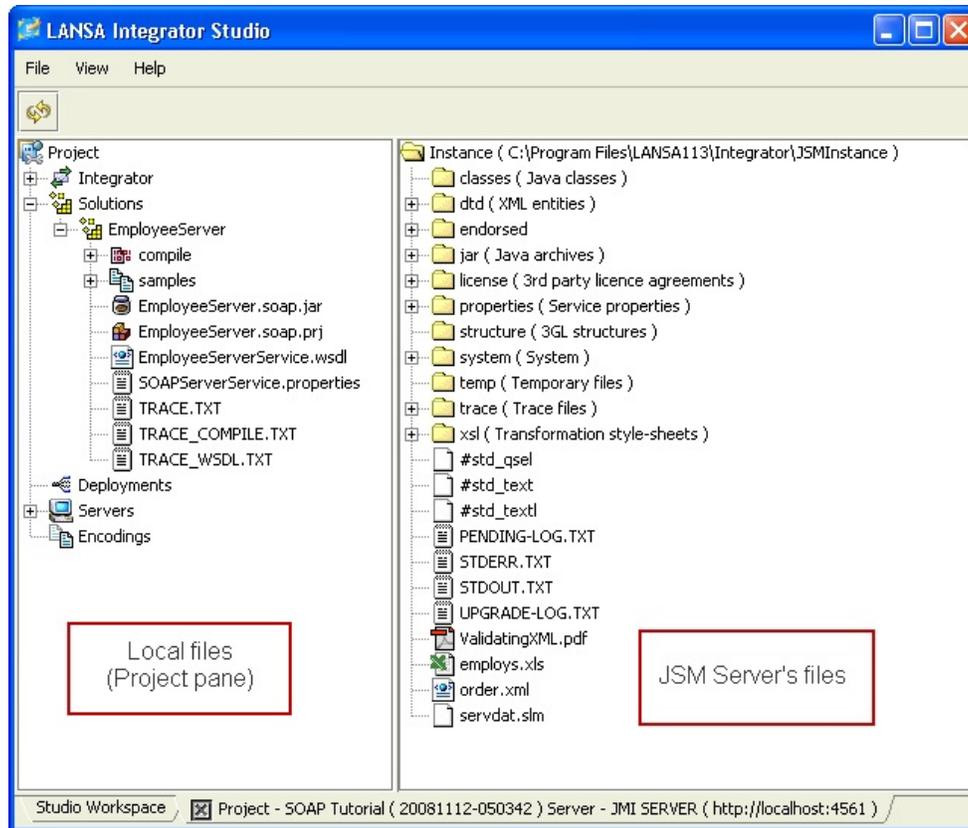
b. If you are using a local web server and JSM Server, your server definition will look like the following:



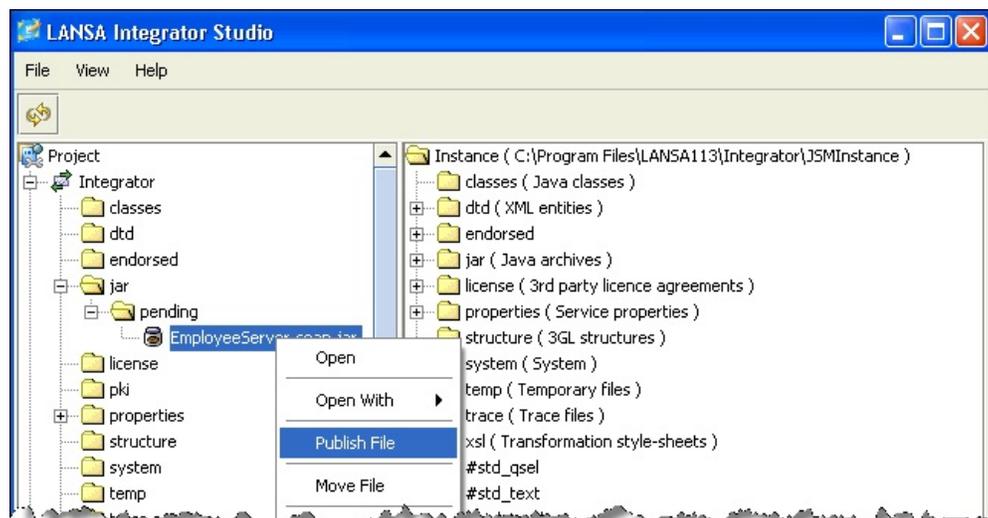
c. Select the appropriate server and use the right mouse menu to Open Server.



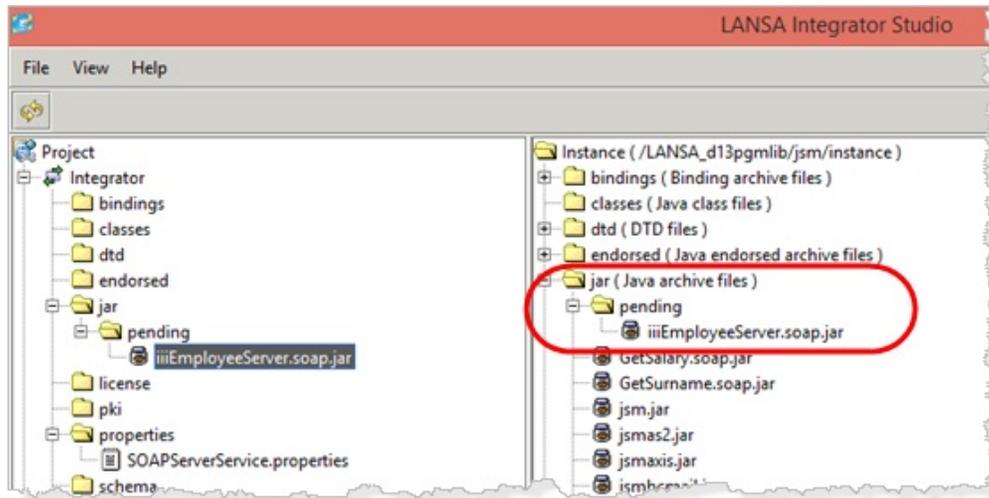
d. Your *Project* tab will now show both your local files and the JSM server's files.



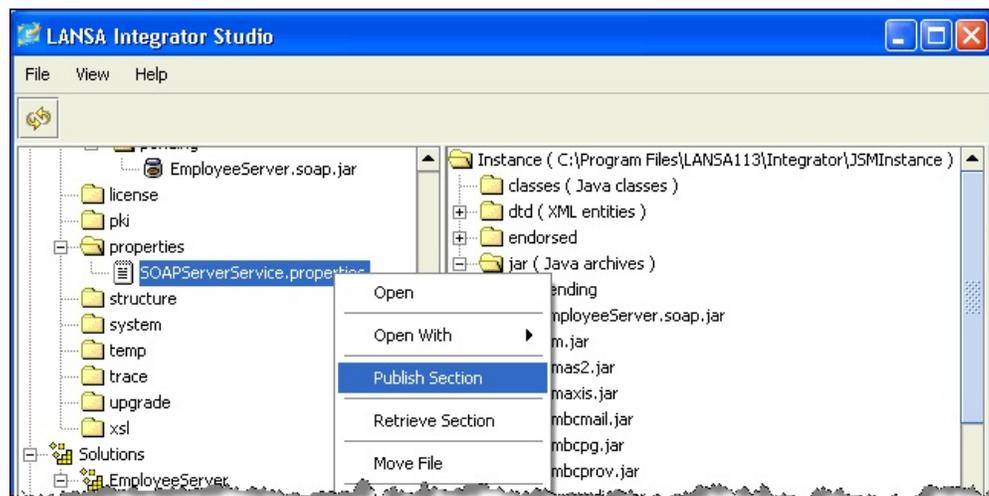
6. Move the jar file to the JSM server:
 - a. Highlight the **jar** file in the *Integrator / jar / pending* folder of the Project panel
 - b. Right click to open the context menu.
 - c. Select *Publish File* from the context menu, and select Yes in the confirm dialog



You will now see the jar files in the *jar / pending* folder of the JSM server.



7. Move the properties file to the JSM server.
 - a. Highlight the properties file in the properties folder of the *Project* pane.
 - b. Right click to open the context menu.
 - c. Select *Publish Section* from the context menu, and select *Yes* in the *Confirm* dialog.

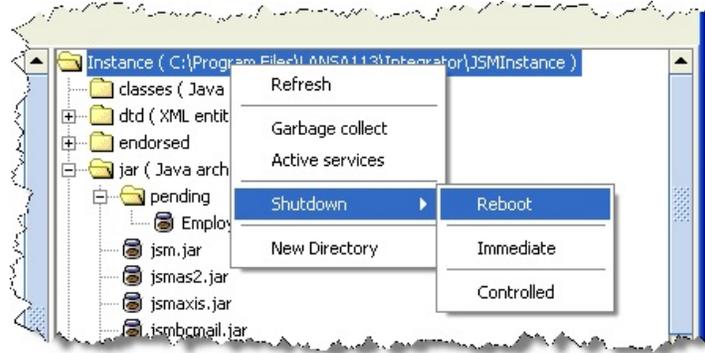


Your properties file is used to add a section to the *SOAPServerService* properties file on the server.

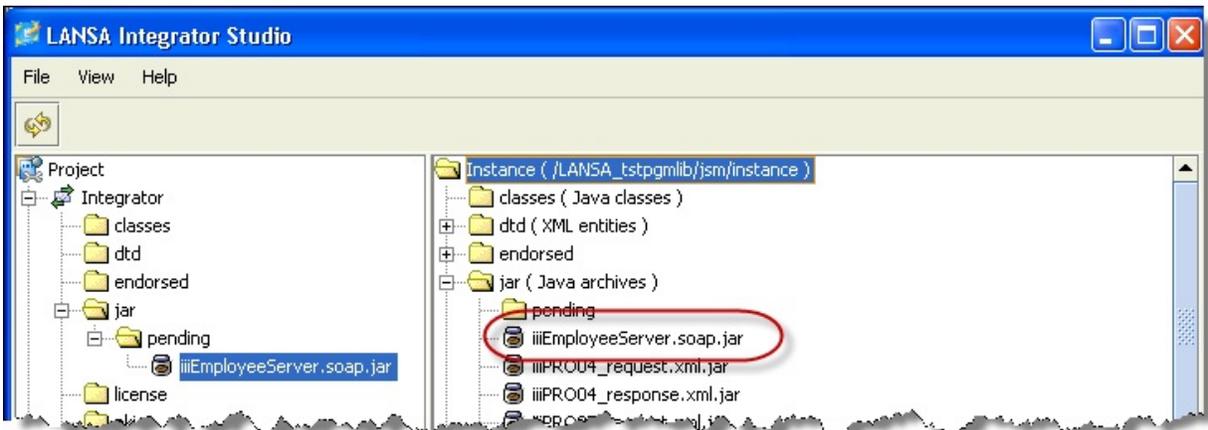
8. You must re-start the JSM Server to move the jar file out of the pending folder.

To do this:

- a. Highlight the *Instance* folder in the JSM server pane
- b. Right click in open the context menu.
- c. Select *Shutdown* and then *Reboot* from the context menu.



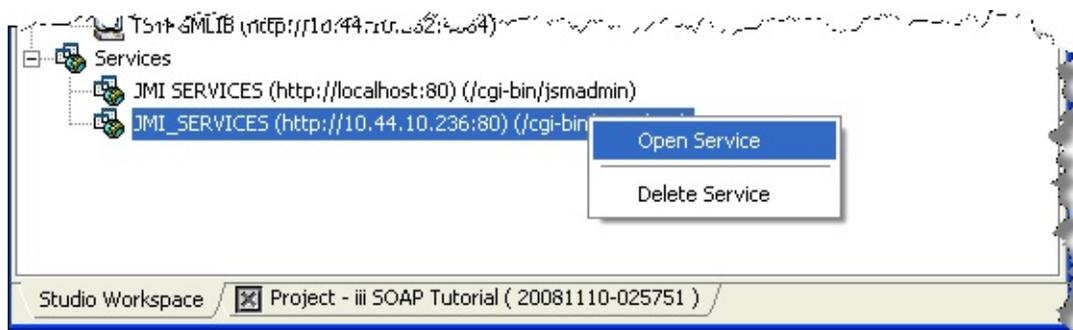
9. When the reboot has been completed, press *F5* to refresh the screen. You will see that the jar files are no longer in the *jar / pending* folder of the JSM server pane.



Part B. Define Service entry

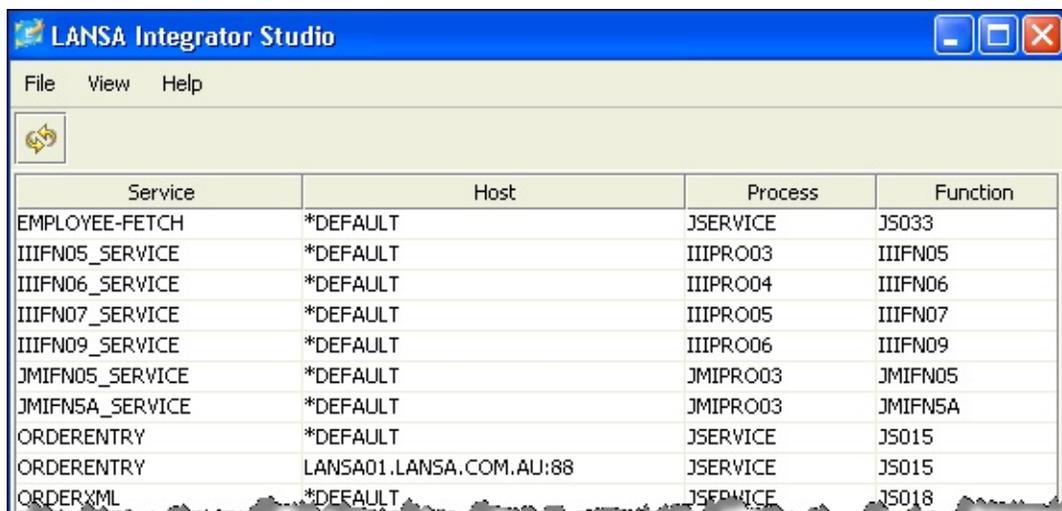
You need to create a service entry that tells JSMDirect which LANSAs function or other program is used to process web service requests for the web service you have defined.

1. Switch to the *Studio Workspace* tab of the LANSAs Integrator Studio.
2. Expand the *Services* folder. You will have defined your *Service* entry in an earlier exercise.
3. Select and right click on the required *Service* to open the context menu.



4. Choose *Open Service* from the context menu.

This opens the *Direct Services* tab which shows the current entries in table DC@W29 for an IBM i JSM Server or file DC_W29.txt for a Windows JSM Server.



The screenshot shows the 'Direct Services' tab in the LANSAs Integrator Studio. The table displays the following data:

Service	Host	Process	Function
EMPLOYEE-FETCH	*DEFAULT	JSERVICE	J5033
IIIFN05_SERVICE	*DEFAULT	IIIPRO03	IIIFN05
IIIFN06_SERVICE	*DEFAULT	IIIPRO04	IIIFN06
IIIFN07_SERVICE	*DEFAULT	IIIPRO05	IIIFN07
IIIFN09_SERVICE	*DEFAULT	IIIPRO06	IIIFN09
JMIFN05_SERVICE	*DEFAULT	JMIPRO03	JMIFN05
JMIFN5A_SERVICE	*DEFAULT	JMIPRO03	JMIFN5A
ORDERENTRY	*DEFAULT	JSERVICE	J5015
ORDERENTRY	LANSA01.LANSA.COM.AU:88	JSERVICE	J5015
ORDERXML	*DEFAULT	JSERVICE	J5018

5. You need to create a new entry so right click in the empty space ... and choose *New* from the context menu to open the *Direct Service Editor*.

The image shows a 'Direct Service Editor' dialog box with the following fields:

- Service: [Empty text box]
- Host: [Empty text box]
- Process: [Empty text box]
- Function: [Empty text box]
- Partition: [Empty text box]
- Language: [Empty text box]
- Program: [Empty text box]
- RDMLX:
- WSDL Path: [Empty text box]

Buttons: OK, Cancel

6. In the *Direct Service Editor*, complete the details for your service entry that will connect your web service. For this tutorial, enter:

Service IIIEMPLOYEESERVER

Host *DEFAULT

Process IIIPRO08

Function IIIFN11

Partition DEM

Language ENG

Program Leave blank. Required if a 3GL program is used for the business function.

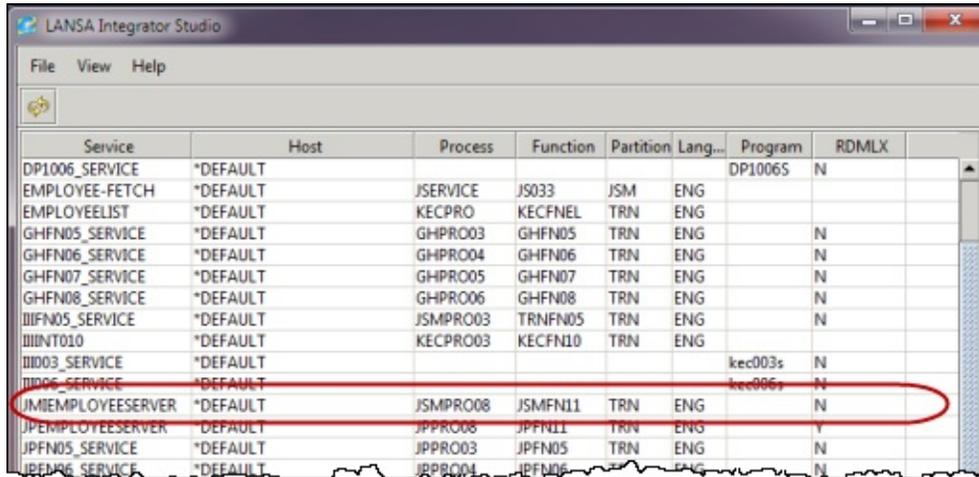
WSDL Leave blank in this case (no external users)

Select the RDMLX checkbox. iiiFN11 will be an RDMLX function and can be executed by X_RUN.

7. When complete, press *OK* to save the *Service* entry details.

8. Press *OK* again.

You will be returned to the *Direct Services* tab where you will see your *Services* entry in the *JSMDirect Services* list.



Service	Host	Process	Function	Partition	Lang...	Program	RDMLX
DP1006_SERVICE	*DEFAULT					DP1006S	N
EMPLOYEE-FETCH	*DEFAULT	JSERVICE	JS033	JSM	ENG		
EMPLOYEELIST	*DEFAULT	KECPRO	KECFNEL	TRN	ENG		
GHFND5_SERVICE	*DEFAULT	GHPRO3	GHFN05	TRN	ENG		N
GHFND6_SERVICE	*DEFAULT	GHPRO4	GHFN06	TRN	ENG		N
GHFND7_SERVICE	*DEFAULT	GHPRO5	GHFN07	TRN	ENG		N
GHFND8_SERVICE	*DEFAULT	GHPRO6	GHFN08	TRN	ENG		N
IIIFN05_SERVICE	*DEFAULT	JSMPRO3	TRNFN05	TRN	ENG		N
IIINT010	*DEFAULT	KECPRO3	KECFN10	TRN	ENG		
IIID03_SERVICE	*DEFAULT					kec003s	N
IIID06_SERVICE	*DEFAULT					kec006s	N
JMIEMPLOYEESEVER	*DEFAULT	JSMPRO8	JSMFN11	TRN	ENG		N
JPEEMPLOYEESEVER	*DEFAULT	JPPRO8	JPFN11	TRN	ENG		Y
JPFN05_SERVICE	*DEFAULT	JPPRO3	JPFN05	TRN	ENG		N
JPFN06_SERVICE	*DEFAULT	JPPRO4	JPFN06	TRN	ENG		N

9. Close the *Direct Services* tab by right clicking and selecting *Close* from the context menu

You have completed the building of your LANSAs Integrator SOAP server solution. What you have built and LANSAs Integrator has generated, is the code necessary to handle all the protocol and format details to receive and respond to web service requests, to unwrap and extract the parameters and to build the SOAP response. You will now build the business logic to support it.

Step 8. Create the SOAP Server Business Logic

Before you can test or use your SOAP Server solution, you need to create the business logic that supports it. That is, a function to receive a request with department code and section code parameters and to build the Employee return parameter which contains the requested employee details.

You do not need to concern yourself with the communication details as LANSAs Integrator and the SOAP Wizard has generated the files which implement that.

Following is an outline of the code's functionality using the LANSAs RDMLX code generated by Integrator Studio as

SAMPLE_RDMLX_GETEMPLOYEES.txt. Note that this program could alternatively have been coded as an RDML function or as an ILE RPG program. Integrator Studio also generates an RPG and an RDML example.

1. The first steps are to create a connection to the JSM server and to load the SOAP Server service. This is accomplished using the JSMX_OPEN Built-In Function and the SERVICE_LOAD service command.

```
* Open service
USE BUILTIN(JSMX_OPEN) TO_GET(#JSMXSTS #JSMXMSG #JSMXHDLE1)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* Load service
CHANGE FIELD(#JSMXCMD)
  TO('SERVICE_LOAD SERVICE(SOAPServerService) SERVICE_CONTENT(*HTTP) TRACE(*YES)
  ')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)
```

2. The function opens the service iiiEMPLOYEESERVER and gets the object *OPERATION. It closes with an error if the operation was not GETEMPLOYEES. Bear in mind that you could develop a function to handle more than one SOAP operation. If the GetEmployees service operation was requested, then the processing proceeds.

```
* Open SOAP service
CHANGE FIELD(#JSMXCMD) TO('OPEN SERVICE(IIIEMPLOYEESERVER)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* Check for operation - GETEMPLOYEES
CHANGE FIELD(#JSMXCMD) TO('GET OBJECT(*OPERATION)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)
-IF COND('#JSMXMSG *NE GETEMPLOYEES')
  * Close SOAP service and send back a SOAP fault message
  USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 'CLOSE FAULT(Incorrect operation)')
  TO_GET(#JSMXSTS #JSMXMSG)
  * Close service
  USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1) TO_GET(#JSMXSTS #JSMXMSG)
  RETURN
-ENDIF
```

- The program uses the GET PARAMETER service command to retrieve the values that were received for the department code and the section code. You will remember that in [Step 5. Map Parameters to Program Variables](#) you mapped these parameters to program variable names, so this call will automatically populate those variables.

```

* Get parameter - DEPARTMENTCODE
CHANGE FIELD(#JSMXCMD) TO('GET PARAMETER(DEPARTMENTCODE) SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* Get parameter - SECTIONCODE
CHANGE FIELD(#JSMXCMD) TO('GET PARAMETER(SECTIONCODE) SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

```

- The program tells the SOAP service that it is beginning to build the SOAP response. This is accomplished using the SET PARAMETER(*RETURN) service command.

The generated code contains a loop since it is aware that the details of each employee will be passed as a fragment. This is the code you need to modify to add the required logic to read employee details from the employee file.

Remember that the employee id, first name, surname and salary fields are already mapped to the elements of the Employee return parameter.

```

* <<< Enter your business logic here >>>

* Set return parameter
CHANGE FIELD(#JSMXCMD) TO('SET PARAMETER(*RETURN)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* <<< Enter the fragment loop logic for EMPLOYEE >>>
-BEGIN_LOOP /* EMPLOYEE */
    * Set fragment loop - EMPLOYEE
    CHANGE FIELD(#JSMXCMD) TO('SET FRAGMENT(EMPLOYEE) SERVICE_EXCHANGE(*FIELD)')
    USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
    EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)
-END_LOOP /* EMPLOYEE */

```

- Finally, the program cleans up by closing the EmployeeServer solution and closing the connection to the JSM server. This will cause the SOAP Server service to send the SOAP response using the data the program has provided.

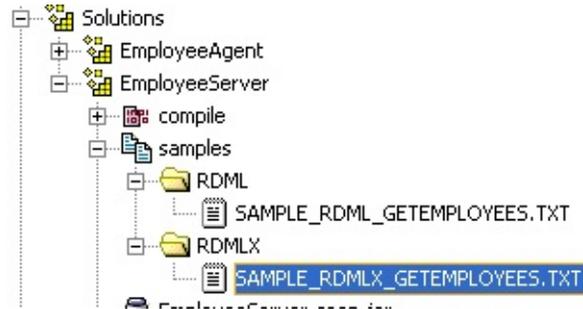
```

* Close SOAP service and send back response
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 'CLOSE') TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

* Close service
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)

```

- Using the LANSAs Editor, create a new process **iiiPRO08 – SOAP Server** and an RDMLX function **iiiFN11 – Get Employees Server**, belonging to it. With your **iii SOAP Tutorial** project open in Integrator Studio, open the generated RDMLX for the EmployeeServer service and copy it into your function (replacing existing code).



As previously, the changes you need to make are very straightforward. Replace the BEGIN_LOOP / END_LOOP with a SELECT/ENDSELECT loop, to retrieve the required fields from the file PSLMST using the correct logical view. Your code should look like the following:

```
* <<< Enter the fragment loop logic for EMPLOYEE >>>
-SELECT FIELDS(#empno #surname #givename #salary) FROM_FILE(pslmst1)
  WITH_KEY(#deptment #section) NBR_KEYS(*compute)

* Set fragment loop - EMPLOYEE
CHANGE FIELD(#JSMXCMD) TO('SET FRAGMENT(EMPLOYEE) SERVICE_EXCHANGE(*FIELD)')
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSMXCMD) TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS #JSMXMSG)
-ENDSELECT
```

Because of the JSMDirect service entry you created in in *Step 7, Part B. Define Service entry*, this function will be automatically invoked by JSMDirect when it receives a matching web service request.

The name of the process and function must be the same as that entered in the *Direct Service* entry.

- Locate the following code and change the service name using your initials:

- * Open SOAP service

Change Field(#JSMXCMD) To('OPEN SERVICE(IIIEMPLOYEESERVER)')

- Compile your function. If you are using an IBM i server, check in to the server and compile.

Summary

Important Observations

- Knowledge of web service standards or protocols were not needed to create this SOAP Server.
- Business logic to process the SOAP Service can be created using either LANSAs RDML, RDMLX functions or a 3GL such as ILE RPG.

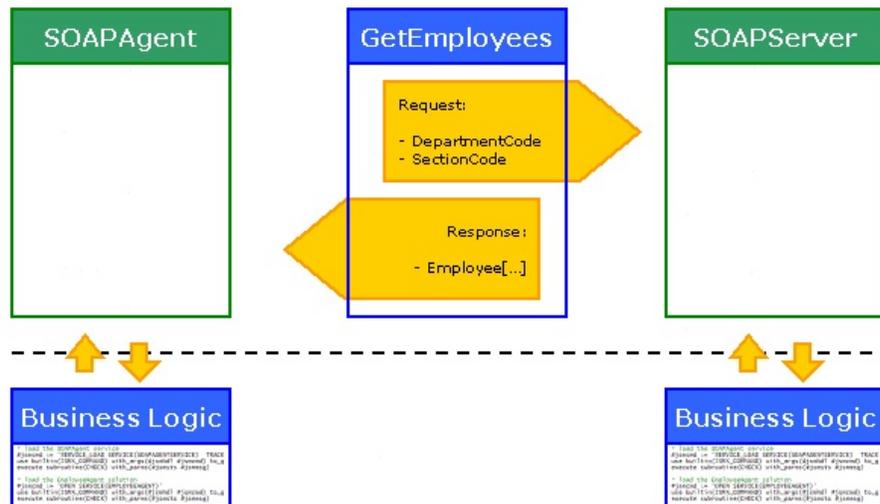
What I Should Know

- The essential steps necessary to use LANSAs Integrator Studio to create a SOAP Server solution that can call or consume a web service, either local or non-local.
- That LANSAs Integrator generates components that handle the complexities of web services standards and protocols.
- How to create a LANSAs function that is called as a SOAP Server web service.

INT010B - SOAP Service - Define Agent

There are three tutorials in this set which show you how to implement a web service using LANSAs Integrator Studio.

The web service created in these tutorials is called **GetEmployees**.



The SOAP Agent consuming the **GetEmployees** web service will pass a department and section to the server. A list of employees for that department and section will be returned by the publisher, the SOAP Server, as an array. If only a department is passed, a list of employees in all sections for that department will be returned. If no department and section is provided then all employees for the company will be returned. The *SOAP Agent* program created to test this web service uses Visual LANSAs, but it could be created using a 3GL such as ILE RPG equally well.

To create the service as shown in the diagram above, you must also complete this tutorial:

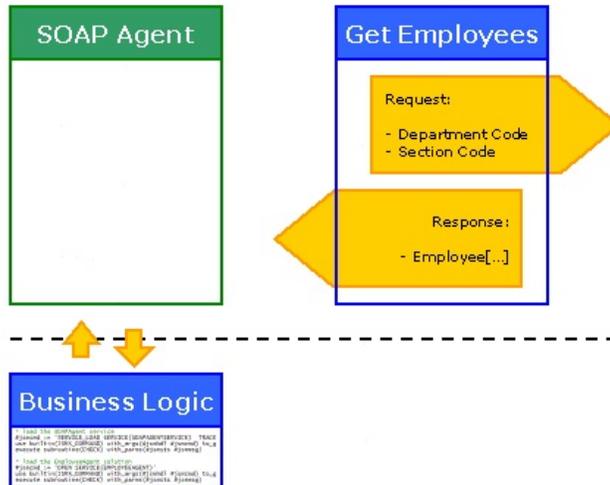
- [INT010A - SOAP Service - Define Server](#)

To test the service, you must complete:

- [INT010C - SOAP Service - Test](#)

Objectives

- To create the Agent side of a web service called **GetEmployees**. In this case, the web service will be the SOAP Server created in *Step 8.Create the SOAP Server Business Logic*.
- To define the SOAP Agent Solution using LANSAs Integrator Studio.



To achieve these objectives, you will complete the following:

- Step 1. Create a New Agent Solution
- Step 2. Map Parameters to Program Variables
- Step 3. Build the SOAP Agent Solution
- Step 4. Deploy the SOAP Agent solution
- Step 5. Review required Program to use the SOAP Agent Solution
- Step 6. Create form iiiFRM02 - Get Employees using SOAP

Step 1. Create a New Agent Solution

In this step you will use the LANSa Integrator Studio to create a new *Solution* in the *Project* that was created for the SOAP Server in INT010A – SOAP Service – Define Server. The new solution will hold the SOAP Agent details.

1. With your **iii SOAP Server** project open in Studio.
2. Select *Solutions*, and select *New Solution* from the context menu to open the *New Solution* dialog box.



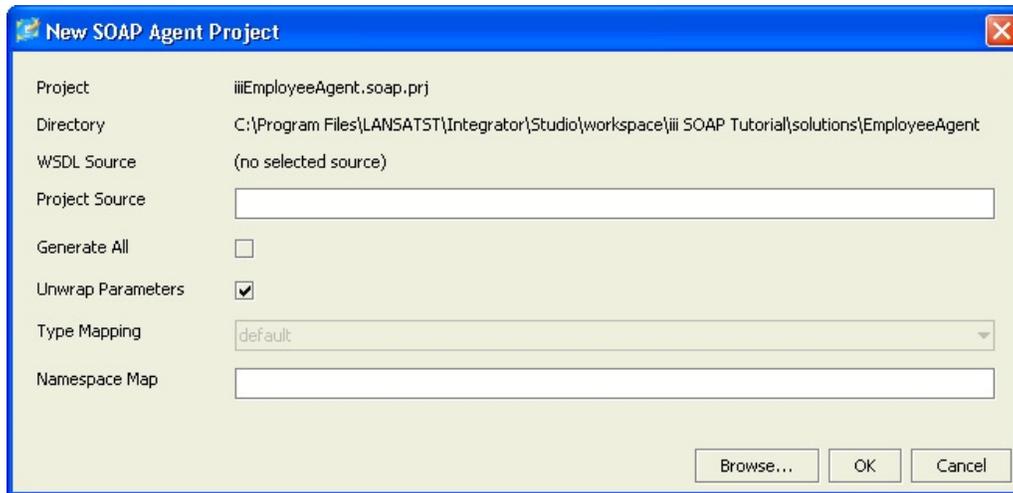
3. In the *New Solutions* dialog box:
 - a. Select the **SOAP Agent Wizard** as the *Tool* from the dropdown list.



- b. Enter the name of the *Group* which is to contain this solution, in this case **EmployeeAgent**. You could select from an existing Group from the dropdown list.
 - c. Enter the name for the solution in *File*, in this case **iiiEmployeeAgent**. This name will be used for the Project File. It will also be used by the SOAP Wizard as a prefix for some of the objects that it generates to support the solution.
6. Confirm that the new Solution Group is to be created



LANSAT Integrator then displays the *New SOAP Agent Project* dialog box.



WSDL Source The fundamental pre-requisite for any SOAP Agent Project is to have the WSDL file that describes the web service you want to use.

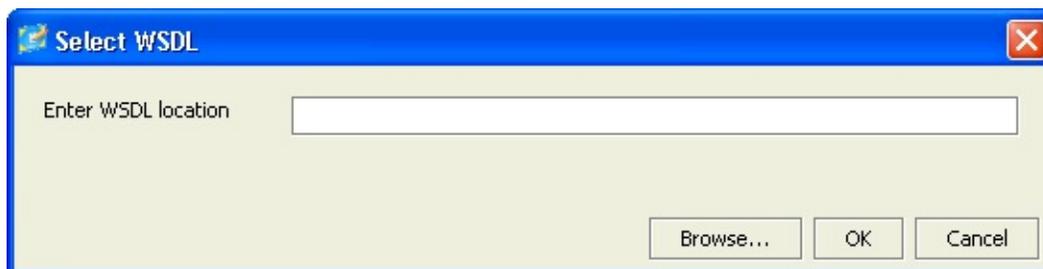
WSDL stands for Web Services Description Language and is a universal language used to describe a web service.

The file, or its location, is provided by the organization that publishes the web service. In this case, the WSDL file was created in INT010A – SOAP Server – Define Server.

7. Locate the *WSDL Source* file.

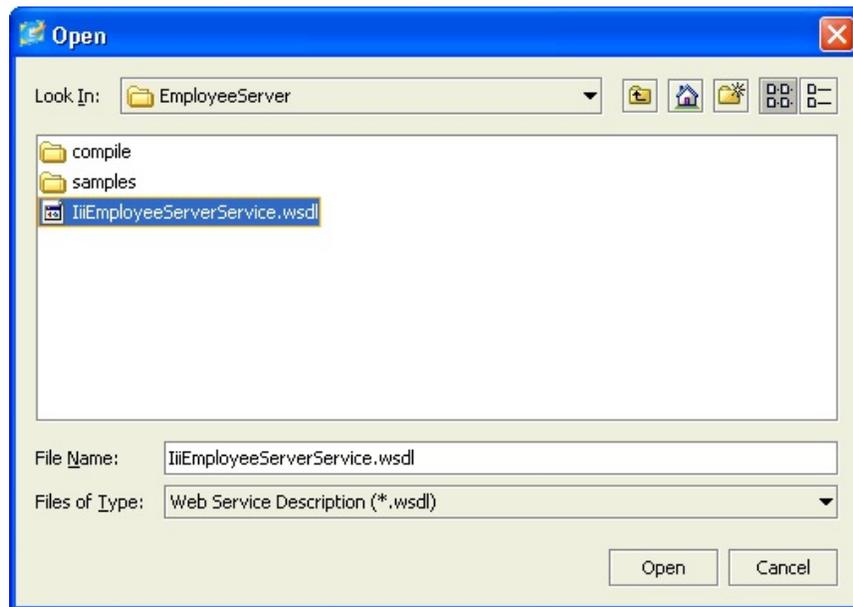
a. Click the *Browse* button of the *New SOAP Agent Project* dialog box.

The *Select WSDL* dialog is opened.

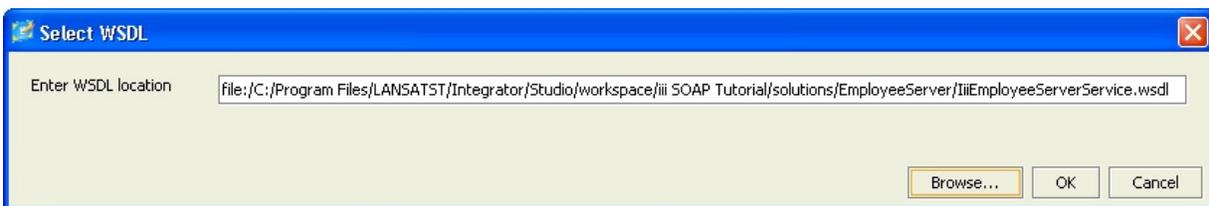


b. If you know the WSDL file's path and name, enter it here otherwise press the *Browse* button to search for it.

The WSDL file will be in the *Solutions / EmployeeServer* folder.

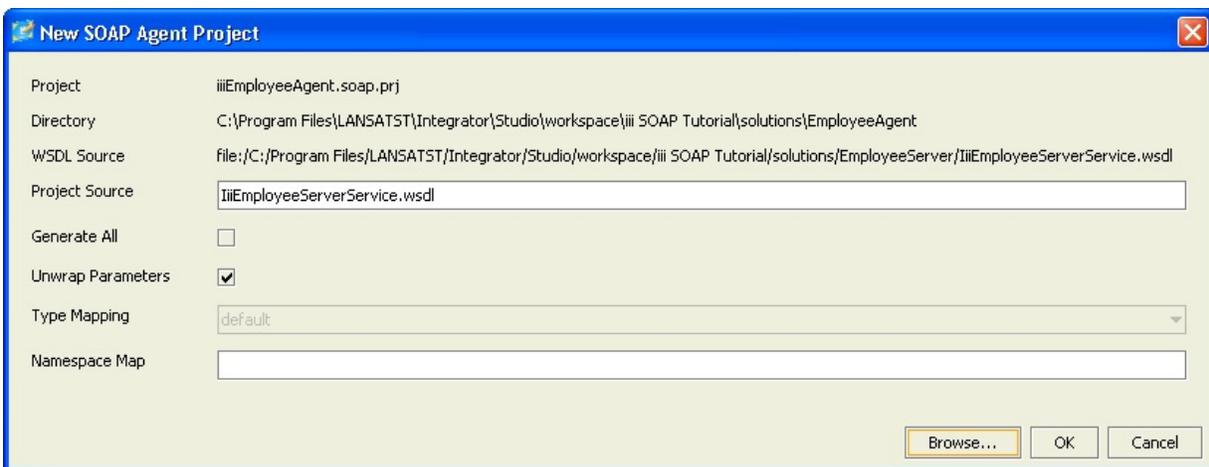


- c. Select the WSDL file and click the *Open* button. You are returned to the *Select WSDL* dialog box.

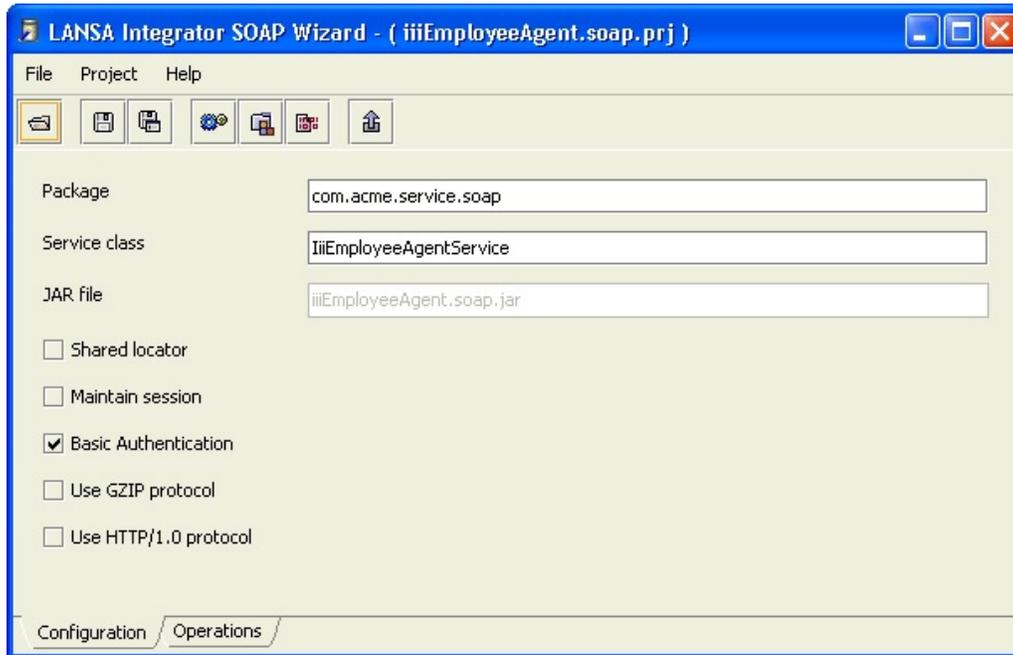


- d. Click *OK*.

You are returned to the *New SOAP Agent Project* dialog where the *WSDL Source* file's path and file name will be displayed.



8. Click *OK* to return to the LANSAs Integrator *SOAP Wizard*.



LANSAs Integrator has filled in the fields of the *SOAP Wizard* dialog with default values.

9. Change the *Package* name.

The *Package* name is the name that the *SOAP Wizard* will use when generating the Java implementation files for your solution, so you need to make it unique to your company. For this exercise use **com.iiicompany.service.soap**.

10. Save your changes and leave the *SOAP Wizard* open.

Proceed to [Step 2. Map Parameters to Program Variables](#).

Step 2. Map Parameters to Program Variables

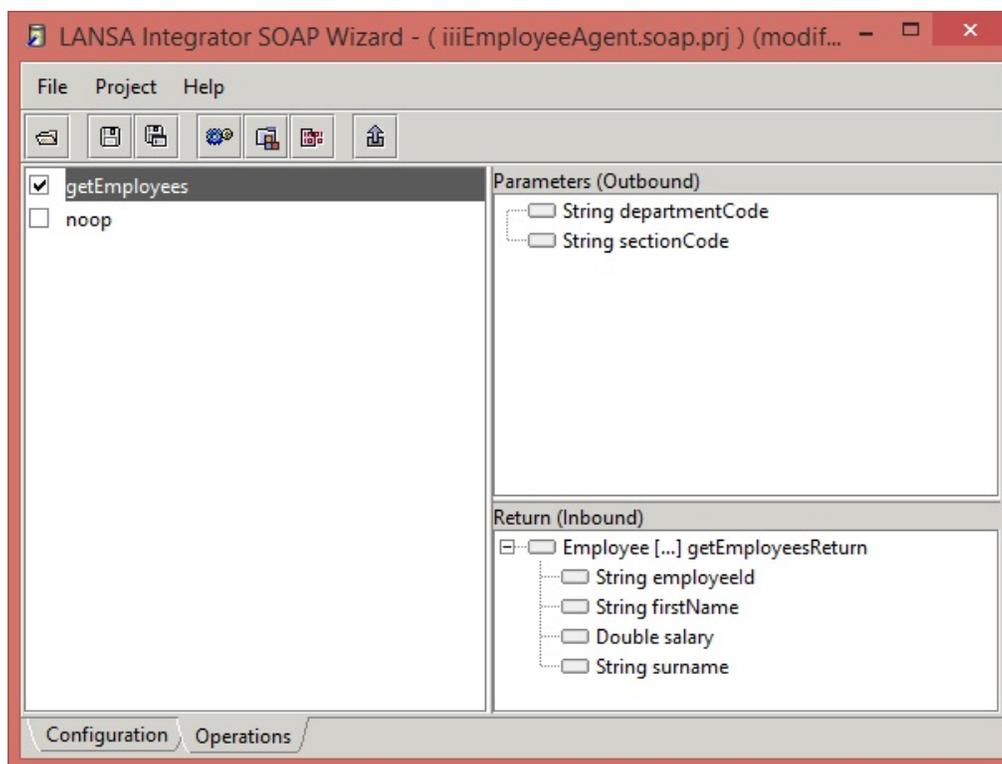
In this step you will map the Outbound and Inbound parameters to the program variable names that you will use in the business logic that calls this web service. The Inbound and Outbound parameters are obtained from the WSDL file you located in the previous step.

1. Switch to the *Operations* tab of the SOAP Wizard.

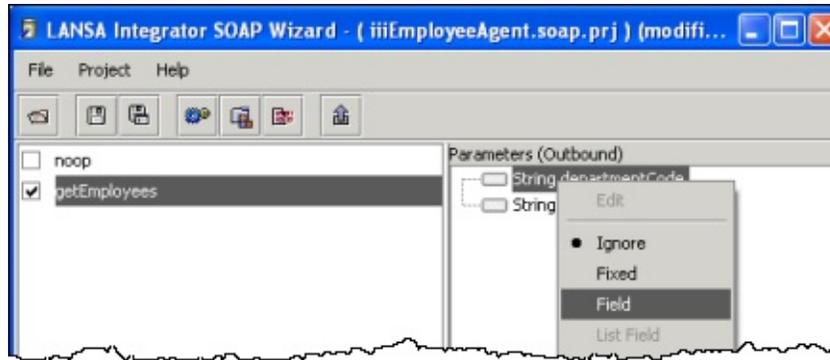
In the *Operations* tab, the left-hand panel shows the names of the web service operations that are defined in the WSDL file which you located in [Step 1. Create a New Agent Solution](#).

2. Tick (P) the box of the Operation that you wish to use. In this case it is **getEmployees**.

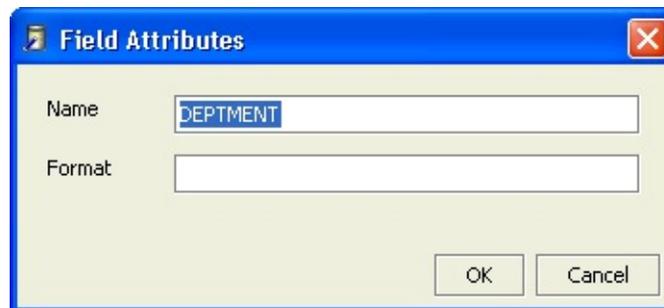
Once ticked and selected, you will see the *Inbound* and *Outbound* parameters you defined in Step 4. Create a new SOAP Operation in INT010A.



3. Map the *Parameters(Outbound)*. These parameters are mapped in exactly the same way as used to map the parameters for EmployeeServer. That is:
 - a. Select and right click the **departmentCode** to open the context menu.

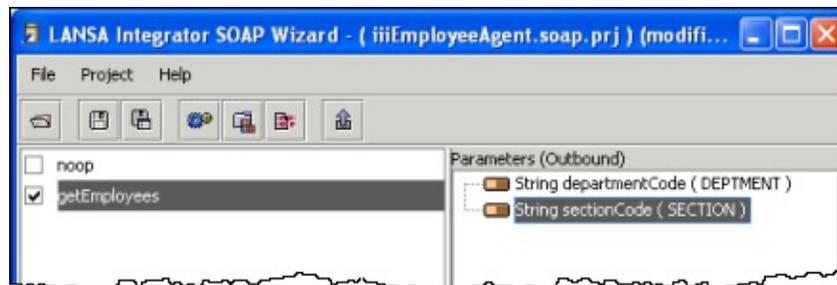


b. Select *Field* from the context menu. The *Field Attributes* dialog box is opened.



c. In the *Field Attributes* dialog box, enter the corresponding program variable name, **DEPTMENT** and press *OK*.

4. Repeat for the **sectionCode** Outbound parameter, mapping it to program variable **SECTION**.

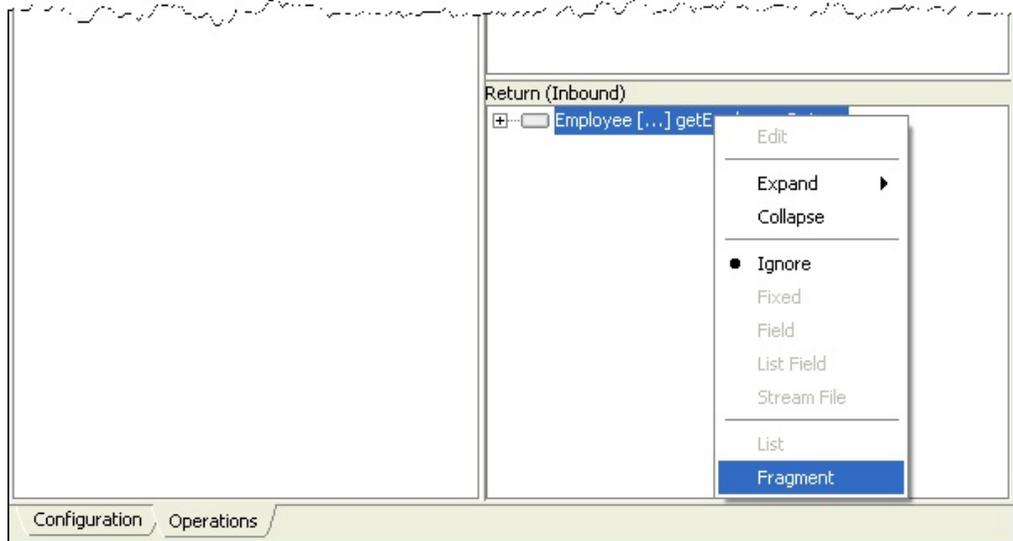


The mapped program variable name is now in brackets following the parameter name.

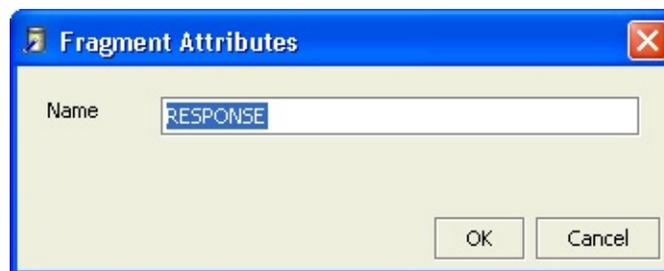
The icon has changed to indicate that the parameter is mapped.

Tip: Instead of right clicking and selecting your action from the context menu, you can press Enter to open whichever parameter is highlighted. Use the Arrow keys to move up and down the list.

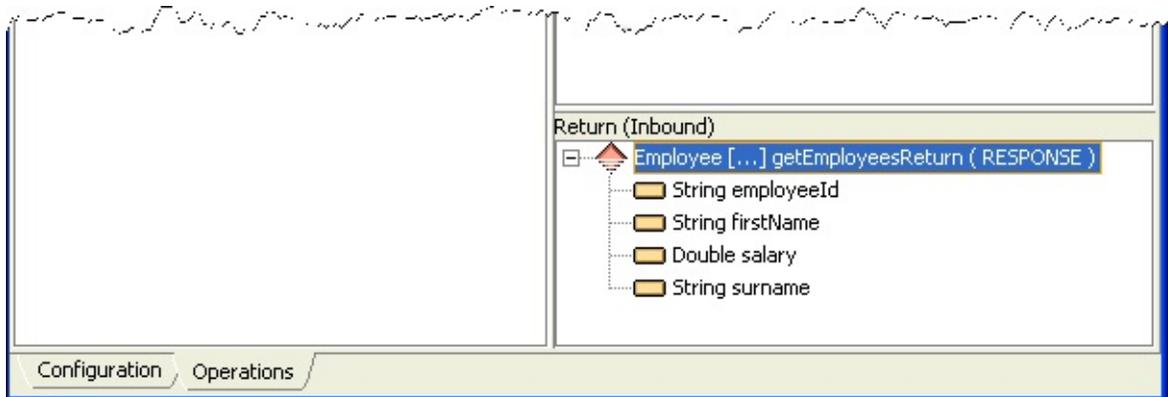
5. Map the *Return (Inbound)* parameters. This parameter is a complex type and because of this, is handled slightly differently to the Outbound Parameters.
- a. Select and right click the *Employee[...]* to display the context menu.



- b. Select **Fragment** from the context menu. This option allows all the elements of Employee to be received in a single call.

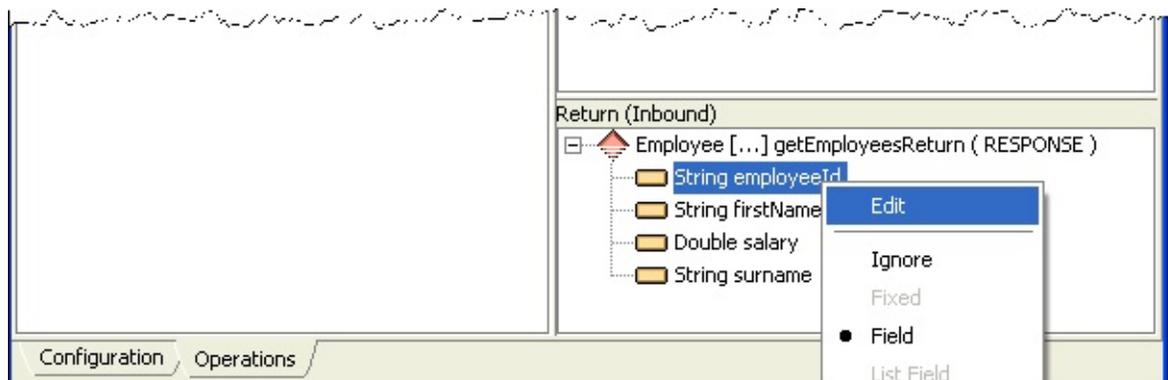


- c. In the *Fragment Attributes* dialog, enter the name of the Fragment, in this case, it is **RESPONSE**.
- d. Press *OK* to confirm this assignment.

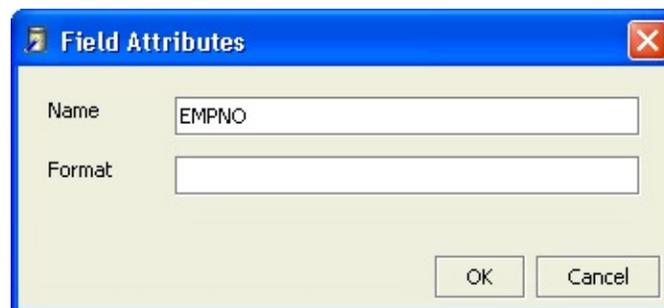


Again you can see the mapped fragment name shown beside the *Employee* Return parameter, and that the icon has changed. In this case, the icon has changed for all the elements of employee because when you marked *Employee* as a Fragment, LANSa Integrator assumed that all the elements would be mapped to fields.

6. You must now map each of the fields in the response.
 - a. Right click on *employeeId* to open the context menu.



- b. Select *Edit* from the context menu to open the *Field Attributes* dialog box.



- c. Enter the Name, in this case, **EMPNO**.
 - d. Leave *Format* blank and Press *OK*.

7. Map each Inbound field to the relevant program variable using these program variable names:

Inbound **Program variable name**

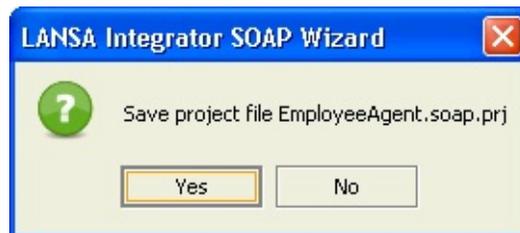
firstName GIVENAME

salary SALARY

surname SURNAME

Note that the program variable used by the Agent program will not normally be the same as the name used for the server's program.

8. When you have assigned all the Inbound fields, Save your work by either pressing the *Save*  button, pressing the *Ctrl + S* keys or use *Save* from the *File* menu.



Step 3. Build the SOAP Agent Solution

In this step you will build the *SOAP Agent* solution and then review the files that LANSAs Integrator has generated.

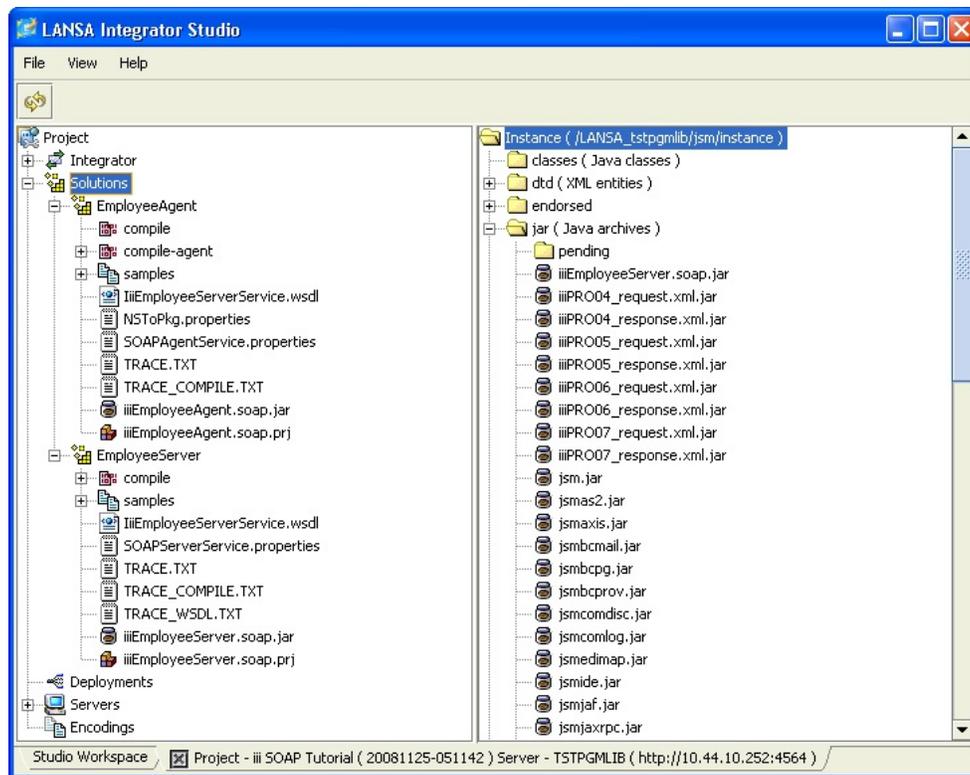
1. With the *Operations* tab still displayed, click the  *Build* button on the toolbar.

When the build has been completed, the following message is displayed.



2. Click *OK* to continue.
3. Exit the *SOAP Wizard*.

You are returned to the LANSAs Integrator Studio. The *Project* tab for your *SOAP Agent* project is displayed.



4. Expand the project *Solutions* folder so that you can see the contents of the

EmployeeAgent Solution.

5. Expand the *EmployeeAgent* folder.

Some of the files that the LANSAs Integrator SOAP Wizard has generated for your Project:

- A **jar** file containing the compiled classes, ready for you to deploy to the web server.
- A **prj** (project) file that you will open if you need to use the SOAP Wizard again to revise what you have done.
- A **properties** file that you will need to deploy to the web server to enable applications to call this web service through your SOAP Agent service.
- Other files containing some sample generated RDML code for an application that uses this web service and trace and log files.

Step 4. Deploy the SOAP Agent solution

You need a server in order to complete this step. If a server is not available, you can complete this step except for Publishing the files.

You have built the SOAP Agent solution in your development environment. Now you need to deploy, or publish, these two generated files to the server:

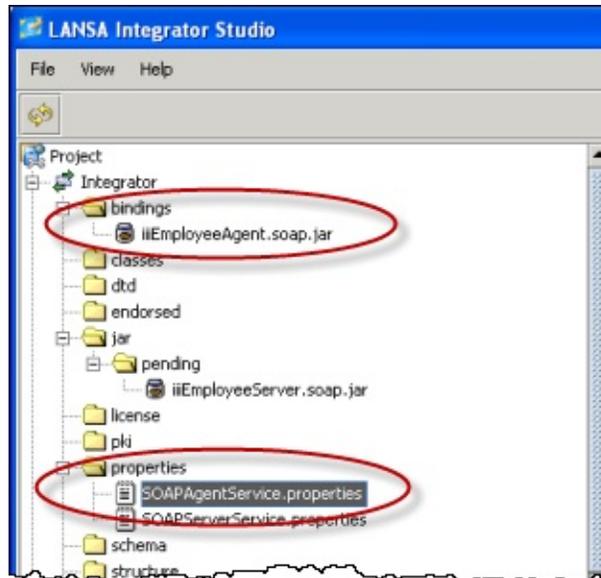
- **iiiEmployeeAgent.soap.jar** file
- **SOAPAgentService.properties** file

The SOAP Agent service is handled like the other Integrator services which you have already implemented in this workshop. When you publish your SOAP Agent files, they are moved to the *Bindings* folder and you will NOT need to reboot the JSM Server to run them.

1. Display the *Project* tab in the *LANSA Integrator studio*.
2. Expand the *EmployeeAgent* folder.
3. Right click the **iiiEmployeeAgent.SOAP.jar** file to open the context menu.
4. From the context menu, select *Send To* and then *Integrator Folder*. Your file will be moved to the *bindings* folder of the Project.
5. Right click on the **SOAPAgentService.properties** file to open the context menu.
6. From the context menu, select *Send To* and then *Integrator Folder*.

Both of your files are now in a folder in the *Project / Integrator* folder.

7. Contract your *Solutions* folder and expand the *Integrator* folder until you can see the **jar** file in the *bindings* folder and the **properties** file that you have just moved to the *properties* folder.



8. Now you must *Publish* your files (that is, move them to the required JSM server). If necessary, in your *Project* tab, open the *Server* you are using, so that the server instance is shown in the right hand pane.
9. Move the **jar** and **properties** files from your local folder to the JSM Server's folder.
 - a. Select and right click the `iiiEmployeeAgent.soap.jar` file in the *bindings* folder to open the context menu.
 - b. Select *Publish File* from the context menu.

Your jar file is now moved to the JSM server's *bindings* folder.
 - c. Right click on the `SOAPAgentService.properties` file in the *Integrator / Properties* folder to open the context menu.
 - d. Choose *Publish Section* from the context menu.

Your properties file is used to add a section to the `SOAPAgentService` properties file on the server.

Step 5. Review required Program to use the SOAP Agent Solution

In this step, you will create the application program that calls the web service using your SOAP Agent solution.

There are two parts to the Agent solution:

- The user based visual form which implements the **getEmployees** SOAP Service application, from the code example supplied in [Step 6. Create form iiiFRM02 - Get Employees using SOAP](#)
- The RDMLX code generated by Integrator Studio for the EmployeeAgent Solution
- The code example is generated as a LANSAs function. You will use the RDMLX version SAMPLE_RDMLX_GETEMPLOYEES.TXT. You could write your own if appropriate.
- Use the generated code, by copying most of it to the *Push Button Click* event routine in the visual form.

Following is a brief description of the essential steps necessary for this click event handler to issue the web service request and receive the results.

The first thing to note is that the LANSAs form communicates with the LANSAs Integrator SOAP service using Built-In Functions specially designed for the purpose.

1. The first steps are to ready the program for calling the Web Service by opening a Java Service Manager connection and loading the SOAPAgent service. The EMPLOYEEAGENT SOAP Agent solution that you have built in this tutorial, is then loaded.

```
* The following fragments are used by the soap binding map
Group_By Name(#RESPONSE) Fields(#EMPNO #GIVENAME #SALARY #SURNAME)

* Open service
Use Builtin(JSMX_OPEN) To_Get(#JSMXSTS #JSMXMSG #JSMXHDLE1)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)

* Load service
Change Field(#JSMXCMD)
  To('SERVICE_LOAD SERVICE(SOAPAgentService) TRACE(*YES)')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
  To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)

* Open SOAP service
Change Field(#JSMXCMD) To('OPEN SERVICE(IIIEMPLOYEEAGENT)')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
  To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)
```

2. The next step calls the GetEmployees operation of the web service. The two SET PARAMETER commands "bind" the program variables to the parameters for the operation. At this point, the user must have input these values. In your form example, the user enters the field value(s) and clicks the Get Employees button, which executes this event routine.

The LANSAs Integrator Java Service Manager has access to the variables in the LANSAs program by means of the SERVICE_EXCHANGE(*FIELD) keyword. It then knows which program field to map to each parameter by means of the mappings that you specified earlier in the SOAP Wizard.

```
* Set parameter - DEPARTMENTCODE
Change Field(#JSMXCMD)
  To('SET PARAMETER(DEPARTMENTCODE) SERVICE_EXCHANGE(*FIELD)')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
  To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)

* Set parameter - SECTIONCODE
Change Field(#JSMXCMD)
  To('SET PARAMETER(SECTIONCODE) SERVICE_EXCHANGE(*FIELD)')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
  To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)
```

3. In the third step, the program calls the web service operation, waits for the response and checks for a null response.

```
* Call remote service
Change Field(#JSMXCMD) To('CALL')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
  To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)

* Check for a possible null return parameter
Change Field(#JSMXCMD) To('IS NOT_NULL(*RETURN)')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
  To_Get(#JSMXSTS #JSMXMSG)
```

4. The fourth step in the generated code is incomplete. Here you will add the business logic. At the moment the program loops, retrieving the RESPONSE fragment until the status (JSMXSTS) is NOFRAGMENT.

```

-If Cond('#JSMXSTS *EQ OK')
  -Begin_Loop /* RESPONSE */
    * Get fragment loop - RESPONSE
    Change Field(#JSMXCMD)
      To('GET FRAGMENT(RESPONSE) SERVICE_EXCHANGE(*FIELD)')
    Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDL1 #JSMXCMD)
      To_Get(#JSMXSTS #JSMXMSG)
    Leave If('#JSMXSTS *EQ NOFRAGMENT')
    Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)
  -End_Loop /* RESPONSE */
-Endif

```

5. Finally, the logic which will form your click event, cleans up by closing the EmployeeAgent solution, unloading the SOAP Agent service and closing the connection to the JSM server.

```

* Close SOAP service
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDL1 'CLOSE')
  To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)

* Unload service
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDL1 'SERVICE_UNLOAD')
  To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)

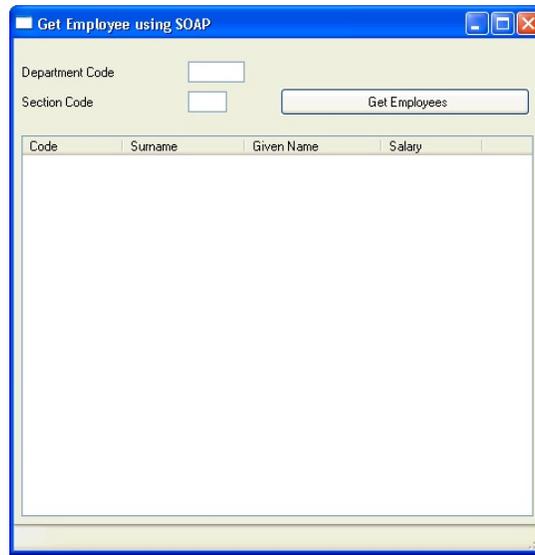
* Close service
Use Builtin(JSMX_CLOSE) With_Args(#JSMXHDL1) To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)

```

Step 6. Create form iiiFRM02 - Get Employees using SOAP

This is a very simple form into which the user enters a department and section code and clicks a button labelled *Get Employees*.

The program then displays a list of employees for that department and section. The visual design of the form is as follows:



1. In the LANSAs Editor, from the *File* menu, create *New / Basic Form*, **iiiFRM02 – Get Employees using SOAP**. Create the form as RDMLX enabled.

Copy and paste the following code into the form:

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(460)
Clientwidth(476) Height(494) Left(261) Top(120) Width(484)
Define Field(#MSGDTA) Type(*CHAR) Length(132)
Override Field(#empno)
Override Field(#surname)
Override Field(#givenname)
Override Field(#salary)
Define_Com Class(#DEPARTMENT.Visual) Name(#IN_DEPT)
Displayposition(1) Height(19) Left(7) Parent(#COM_OWNER)
Tabposition(1) Top(21) Usepicklist(False) Width(201)
Define_Com Class(#SECTION.Visual) Name(#IN_SECT) Displayposition(2)
Height(19) Left(7) Parent(#COM_OWNER) Tabposition(2) Top(48)
Usepicklist(False) Width(185)
```

```

Define_Com Class(#PRIM_PHBN) Name(#phbnGetEmployees) Caption('Get
Employees') Displayposition(3) Left(240) Parent(#COM_OWNER)
Tabposition(3) Top(45) Width(225)
Define_Com Class(#PRIM_LTVW) Name(#empList) Componentversion(2)
Displayposition(4) Fullrowselect(True) Height(345) Left(7)
Parent(#COM_OWNER) Showsortarrow(True) Tabposition(4) Top(88)
Width(461)
Define_Com Class(#PRIM_LVCL) Name(#empList_1) Caption('Code')
Captiontype(Caption) Displayposition(1) Parent(#empList) Source(#EMPNO)
Define_Com Class(#PRIM_LVCL) Name(#empList_2) Caption('Surname')
Captiontype(Caption) Displayposition(2) Parent(#empList)
Source(#SURNAME) Width(24)
Define_Com Class(#PRIM_LVCL) Name(#empList_3) Caption('Given
Name') Captiontype(Caption) Displayposition(3) Parent(#empList)
Source(#GIVENAME) Width(27)
Define_Com Class(#PRIM_LVCL) Name(#empList_4) Caption('Salary')
Captiontype(Caption) Displayposition(4) Parent(#empList)
Source(#SALARY)
Define_Com Class(#PRIM_STBR) Name(#STBR_1) Displayposition(5)
Height(24) Left(0) Messageposition(1) Parent(#COM_OWNER)
Tabposition(5) Tabstop(False) Top(436) Width(476)

```

```

Evtroutine Handling(#com_owner.Initialize)
Set Com(#com_owner) Caption(*component_desc)
Endroutine

```

*

```

Evtroutine Handling(#phbnGetEmployees.Click)

```

```

Endroutine

```

```

* -----

```

```

* Subroutine CHECK - check the JSM status after a JSM call and handle

```

```

* .           exceptional conditions

```

```

* -----

```

```

Subroutine Name(CHECK) Parms((#jsmxsts *RECEIVED) (#jsmxmsg
*RECEIVED))

```

```

If (#jsmxsts *NE OK)

```

```

* Close service and send the HTTP response

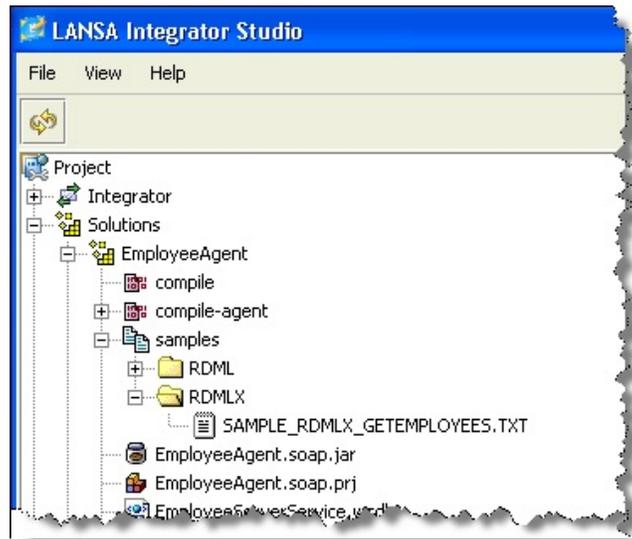
```

```

Use Builtin(JSMX_CLOSE) With_Args(#jsmxhdl1) To_Get(#jsmxsts
#jsmxmsg)
#MSGDTA := 'Error Status Code: ' + #JSMXSTS
Message Msgid(DCM9899) Msgf(DC@M01) Msgdta(#MSGDTA)
#MSGDTA := 'Error Message: ' + #JSMXMSG
Message Msgid(DCM9899) Msgf(DC@M01) Msgdta(#MSGDTA)
Endif
Endroutine
End_Com

```

2. In Integrator Studio open your *iii SOAP Tutorial* project. Locate the RDMLX code generated for your *EmployeeAgent* solution and open it the text editor. See file *SAMPLE_RDMLX_GETEMPLOYEES.TXT*.



3. Copy all of the code between:

- * The following fields are used by the soap binding map
- * #DEPARTMENT
-

and

- * Close service

```

USE      BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDL1)
TO_GET(#JSMXSTS #JSMXMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMXSTS
#JSMXMSG)

```

including the code shown, into the event routine handling

#phbnGetEmployees.Click in your form. This will provide most of the logic needed to call the SOAP service as described in the previous step.

4. Examine the definition of visual fields DEPARTMENT and SECTION on the form and note that they are named IN_DEPT and IN_SECT. The soaptutorial service maps these fields as variables DEPARTMENT and SECTION. You need to set the value of DEPARTMENT and SECTION variables before the set parameter occurs. Locate the comment lines for * set parameter – DEPARTMENTCODE and * set parameter – SECTIONCODE and add an assign statement to do this. Your code should now look like the following:

```
* Set parameter - DEPARTMENTCODE
#department := #in_dept
Change Field(#JSMXCMD)
  To('SET PARAMETER(DEPARTMENTCODE) SERVICE_EXCHANGE(*FIELD)')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
  To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)

* Set parameter - SECTIONCODE
#section := #in_sect
Change Field(#JSMXCMD)
  To('SET PARAMETER(SECTIONCODE) SERVICE_EXCHANGE(*FIELD)')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
  To_Get(#JSMXSTS #JSMXMSG)
Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)
```

5. Locate the * Open SOAP Service comment line. Change the following line to use your SOAP service name:

* Open SOAP Service

Change Field(#JSMXCMD) To('OPEN
SERVICE(**IIIEMPLOYEEAGENT**)')

6. In your form, locate the following code:

```
-If Cond('#JSMXSTS *EQ OK')
  -Begin_Loop /* RESPONSE */
    * Get fragment loop - RESPONSE
    Change Field(#JSMXCMD)
      To('GET FRAGMENT(RESPONSE) SERVICE_EXCHANGE(*FIELD)')
    Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
      To_Get(#JSMXSTS #JSMXMSG)
    Leave If('#JSMXSTS *EQ NOFRAGMENT')
    Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)
  -End_Loop /* RESPONSE */
-Endif
```

The command GET FRAGMENT(RESPONSE)

SERVICE_EXCHANGE(*FIELD) will map the returned fields into your program variables. This means that within this loop, you can add entries to the list view on the form. Your completed code should look like the following:

```

-If Cond('#JSMXSTS *EQ OK')
  Clr_List Named(#emplist)
  -Begin_Loop /* RESPONSE */
    * Get fragment loop - RESPONSE
    Change Field(#JSMXCMD)
      To('GET FRAGMENT(RESPONSE) SERVICE_EXCHANGE(*FIELD)')
    Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
      To_Get(#JSMXSTS #JSMXMSG)
    -Leave If('#JSMXSTS *EQ NOFRAGMENT')
    Execute Subroutine(CHECK) With_Parms(#JSMXSTS #JSMXMSG)
    Add_Entry To_List(#emplist)
  -End_Loop /* RESPONSE */
-Endif
```

7. Compile your form.

Summary

Important Observations

- Knowledge of web services standards and protocols is not required to create this SOAP Agent.
- LANSAs RDML or RDMLX code or a 3GL such as ILE RPG may be used to create the business logic used by this SOAP Agent.

Tips & Techniques

- This example used fragments for a simple list, because no simple field values were being returned. When field values are returned as well as a simple list, the recurring entries can be defined as a list and the results are mapped into a working list or 3GL data structure.
- The supplied code turns on tracing at the service level. You should turn this off for deployment to a production system.

What I Should Know

- The essential steps necessary to use LANSAs Integrator Studio to implement a SOAP Agent solution – that is, a solution that calls or consumes a web service.
- That LANSAs Integrator generates components that handle the complexities of web services standards and protocols.
- How to create a Visual LANSAs application program that uses a SOAP Agent solution to call the web service.

INT010C - SOAP Service - Test

When you begin this tutorial, the third in the SOAP set, you have built your LANSAs Integrator SOAP Agent and SOAP Server Solutions and your solution and business logic is complete. All that remains is to test the solution.

What you have built in this tutorial, is a solution that allows a business application to call a web service operation and receive results from it without having to concern itself with any of the web service protocols and standards. The technical details have been taken care of by the LANSAs Integrator SOAP Agent service and by the code generated by the SOAP Wizard.

As an application developer, you did not need to concern yourself with these details as LANSAs Integrator and the files the SOAP Wizard has generated, take care of this.

In *INT010B SOAP Service - Define Agent*, you developed a simple form which you can use to test your SOAP Agent solution. In this form, the user enters a department code and section code and clicks a button. Then the program displays a list of employees for that department.

Objectives

- To test the SOAP Server and Agent Solutions created in the two preceding tutorials.

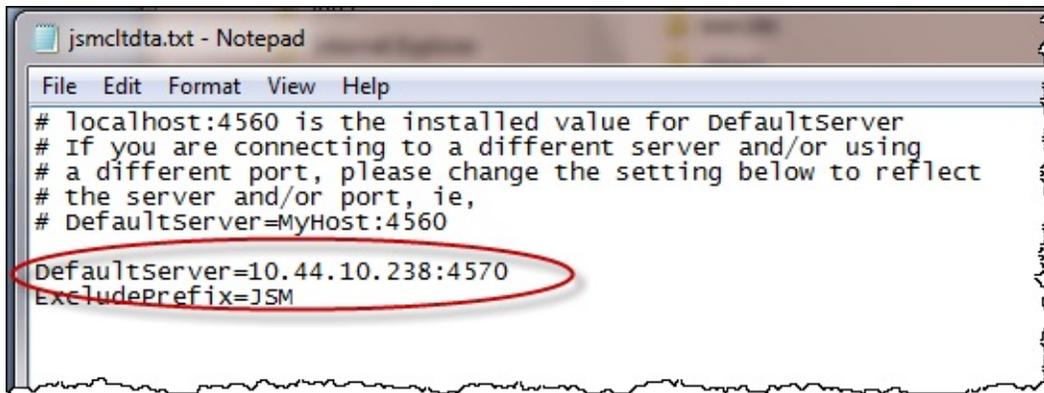
In order to complete this tutorial, you must have completed:

- [INT010A - SOAP Service - Define Server](#)
- [INT010B - SOAP Service - Define Agent](#)

Note: To run this test you should either be:

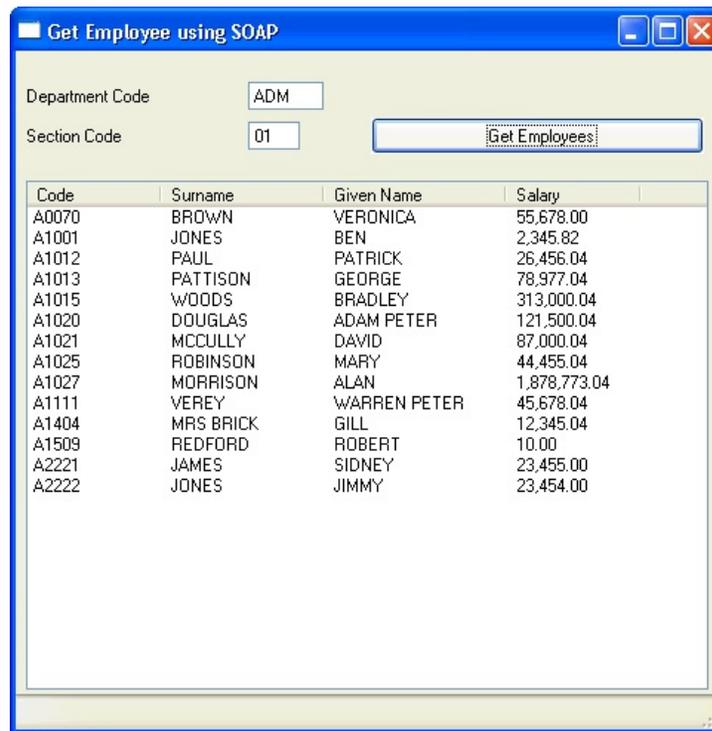
- running a local web server and JSM Server and have your server function compiled locally
- or if using an IBM i JSM Server, have your server function checked in and compiled on the server and the local JSM "data area" file **jsmcltdta.txt** configured to specify your IBM i server as the default JSM Server. See your equivalent of this path for this file:

C:\Program Files\LANSAs\X_WIN95\X_LANSAs



```
jsmcltdta.txt - Notepad
File Edit Format View Help
# localhost:4560 is the installed value for DefaultServer
# If you are connecting to a different server and/or using
# a different port, please change the setting below to reflect
# the server and/or port, ie,
# DefaultServer=MyHost:4560
DefaultServer=10.44.10.238:4570
ExcludePrefix=JSM
```

1. In Visual LANSA, locate, on the *Favorites* tab, the form that you created in *Step 6.Create Form IIIFRM02 –Get Employees using SOAP* of *INT010B*.
2. Right click on the form and select *Execute* to run it.
3. Enter the Department Code, **ADM** and section code, either **01** or **02** and press the *Get Employees* button
4. The response should be something like this:



Code	Surname	Given Name	Salary
A0070	BROWN	VERONICA	55,678.00
A1001	JONES	BEN	2,345.82
A1012	PAUL	PATRICK	26,456.04
A1013	PATTISON	GEORGE	78,977.04
A1015	WOODS	BRADLEY	313,000.04
A1020	DOUGLAS	ADAM PETER	121,500.04
A1021	MCCULLY	DAVID	87,000.04
A1025	ROBINSON	MARY	44,455.04
A1027	MORRISON	ALAN	1,878,773.04
A1111	VEREY	WARREN PETER	45,678.04
A1404	MRS BRICK	GILL	12,345.04
A1509	REDFORD	ROBERT	10.00
A2221	JAMES	SIDNEY	23,455.00
A2222	JONES	JIMMY	23,454.00

5. Check your results are as expected, when you leave both department and section blank, and when only department is specified.
- 6.If errors occur, remember to use Integrator Studio to open the server and review the trace files. You could also run the form in debug mode, if

necessary.

You have now successfully completed the building of your LANSAs Integrator SOAP Server & Agent Solutions. What you have built and LANSAs Integrator has generated is the code necessary to handle all the protocol and format details of receiving and responding to web service requests, of unwrapping and extracting the parameters and of building the SOAP response.

INT011 - Create Excel Document

Objectives

- To create an RDMLX function which is passed department and section code and creates an Excel document containing a list of employees for the section, using the ExcelReadService.
- To create an RDML function to handle the user interface and call the RDMLX function.

To achieve these objectives you will complete the following:

- [Step 1. Create RDMLX function iiiFN12](#)
- [Step 2. Complete function iiiFN12](#)
- [Step 3. Create RDML function iiiFN13](#)
- [Summary](#)

Before You Begin

You must have Microsoft Excel or the Excel Viewer (a free download from Microsoft) installed to complete this exercise.

Step 1. Create RDMLX function iiiFN12

This is an advanced exercise and less detailed instructions have been provided.

1. Create a new process **iiiPRO09 – Excel Example**. Create an RDMLX function **iiiFN12 – Create Employee Excel**, using template JSMXSKELE for the service ExcelReadService.
2. As you have seen in an earlier exercise, this template uses a subroutine KEYWRD to construct the field JSMXCMD. Using RDMLX coding techniques this approach is not really needed, since one statement (ASSIGN command) can create the complicated string needed for each JSM command (JSMXCMD).
3. Delete the KEYWRD subroutine. Change the SERVICE_LOAD logic to create the command using an ASSIGN command. e.g.

```
* BUILD THE SERVICE LOAD COMMAND
#jsmxcmd := 'SERVICE_LOAD SERVICE(ExcelReadService)'
USE BUILTIN(jsmx_command) WITH_ARGS(#jsmxhdl1 #jsmxcmd)
TO_GET(#jsmsts #jsmmsg)
EXECUTE SUBROUTINE(CHECK_STS)
```

Remove unnecessary code. Each of your JSMX_command built-in function executions should use this structure.
4. Define a working list named EMPLOYEES, for employee data containing EMPNO, GIVENAME, SURNAME, STARTDTER, PHONEHME, and SALARY. Allow up to 100 entries.
5. Refer to the *LANSA Integrator guide / Java Services Manager Services / ExcelReadService*. Having read these notes you should know that once your function has built a list of employees, it needs to do the following:
 - Open the Excel file using the OPEN command with FILE and MODE parameters.
 - Write to the Excel file using the WRITE command with SHEET and R1C1 parameters. The working list name is passed on the JSMX_COMMAND. **Note:** SHEET(Sheet1) and R1C1(1,1) are defaults.
 - Close the Excel file using the CLOSE command.

Step 2. Complete function iiiFN12

1. In the section of the function containing the comment YOUR OWN LOGIC HERE, add code to clear the employees list, select from the logical file PSLMST1 with keys DEPARTMENT and SECTION, number of keys *COMPUTE. Add entries to the employees list.
2. Function iiiFN12 will be called by an RDML functions which checks the requested department and section are valid and that employees exist for this section. This error checking will not be required in iiiFN12.
3. To create the Excel file, you need to use the OPEN command using *WRITE mode. Create the OPEN command string (JSMXCMD) and execute it using the JSMX_COMMAND BIF:

The file parameter of the command needs to define the file path (within the JSM instance), file name, and ensure that a new file is created each time (instead of overwriting one file repeatedly). Create the file in the /training subdirectory, with a name beginning iiiemp. Append the current date and time to the file name. The field DATETIMEC contains this value as a character value. The directory must exist.

For a real application you may need to consider how to separate the same output files created by different users.

Review *ExcelReadService* in the *LANSA Integrator* guide for details. You can open this guide directly from *Help / Services* in *Integrator Studio*.

4. Create the WRITE command string to write to Sheet1 in position Row 1, Cell 1. Execute it with the JSMX_COMMAND BIF. Since the sheet and row/cell parameters are both defaults, they do not need to be specified.

Note: In this case the TO_GET parameter must include the name of the employees working list.

5. Close the Excel file using the CLOSE command via the JSMX_COMMAND BIF. **Note:** This is the **ExcelReadService CLOSE** command, not the JSMX_CLOSE command.
6. Remember to include code to execute the CHECK_STS subroutine each time you execute the JSMX_COMMAND BIF.
7. At the end of your logic, if the JSM status (field JSMSTS) is OK, issue a message that the Excel file was created. Note that with RDMLX coding

techniques you can easily add the generated Excel file name into this message.

8. Your logic should look like the following:

```
FUNCTION OPTIONS(*DIRECT)
DEF_LIST NAME(#employs) FIELDS(#empno #givenname #surname
#STARTDTER #PHONEHME #salary) TYPE(*working) ENTRYS(100)
* OPEN JSM AND VERIFY STATUS
USE BUILTIN(jsmx_open) TO_GET(#jsmsts #jsmmsg #jsmxhdle1)
EXECUTE SUBROUTINE(CHECK_STS)
* BUILD THE SERVICE LOAD COMMAND
#jsmxcmd := 'SERVICE_LOAD SERVICE(ExcelReadService)'
USE BUILTIN(jsmx_command) WITH_ARGS(#jsmxhdle1 #jsmxcmd)
TO_GET(#jsmsts #jsmmsg)
EXECUTE SUBROUTINE(CHECK_STS)
* YOUR OWN LOGIC HERE
CLR_LIST NAMED(#employs)
SELECT FIELDS(#employs) FROM_FILE(pslmst1) WITH_KEY(#deptment
#section) NBR_KEYS(*compute)
ADD_ENTRY TO_LIST(#employs)
ENDSELECT
* Create Excel File
#jsmxcmd := 'OPEN FILE(training/iiiemp' + #datetimeec + '.XLS)
MODE(*WRITE)'
USE BUILTIN(jsmx_command) WITH_ARGS(#jsmxhdle1 #jsmxcmd)
TO_GET(#jsmsts #jsmmsg)
EXECUTE SUBROUTINE(CHECK_STS)
* Write to Excel file
#jsmxcmd := 'WRITE'
USE BUILTIN(jsmx_command) WITH_ARGS(#jsmxhdle1 #jsmxcmd)
TO_GET(#jsmsts #jsmmsg #employs)
EXECUTE SUBROUTINE(CHECK_STS)
* Close Excel
USE BUILTIN(jsmx_command) WITH_ARGS(#jsmxhdle1 CLOSE)
TO_GET(#jsmsts #jsmmsg)
EXECUTE SUBROUTINE(CHECK_STS)
IF (#jsmsts = OK)
MESSAGE MSGTXT('Excel file ' + 'iiiemp' + #datetimeec + '.xls produced')
ENDIF
```

```

* UNLOAD SERVICE
USE BUILTIN(jsmx_command) WITH_ARGS(#jsmxhdl1
SERVICE_UNLOAD) TO_GET(#jstmsts #jstmmsg)
EXECUTE SUBROUTINE(CHECK_STS)
* CLOSE JSM AND VERIFY STATUS
USE BUILTIN(jsmx_close) WITH_ARGS(#jsmxhdl1) TO_GET(#jstmsts
#jstmmsg)
EXECUTE SUBROUTINE(CHECK_STS)
RETURN
*
SUBROUTINE NAME(CHECK_STS)
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132)
IF COND(#JSMSTS *NE OK')
USE BUILTIN(BCONCAT) WITH_ARGS('Error Status Code: ' #JSMSTS)
TO_GET(#MSGDTA)
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
USE BUILTIN(BCONCAT) WITH_ARGS('Error Message: ' #JSMMSG)
TO_GET(#MSGDTA)
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
ENDIF
ENDROUTINE

```

9. Compile function iiiFN12. If you are using the JSM server on your IBM i, check in and compile function iiiFN12.

Step 3. Create RDML function iiiFN13

This function will accept the department and section code, check they are valid and call function **iiiFN12**.

1. Create an RDML function **iiiFN13 – Call Create Excel**, belonging to process **iiiPRO09**. Do not use a template. Note this is not an RDMLX enabled function.
2. Write your function based on the following pseudo code:
 - Loop until exit or cancel key
 - Request fields DEPARTMENT and SECTION
 - Use BEGINCHECK loop and FILECHECK to validate these fields against file SECTAB
 - Exchange fields DEPARTMENT and SECTION and call function iiiFN12
3. Your completed code should look like the following

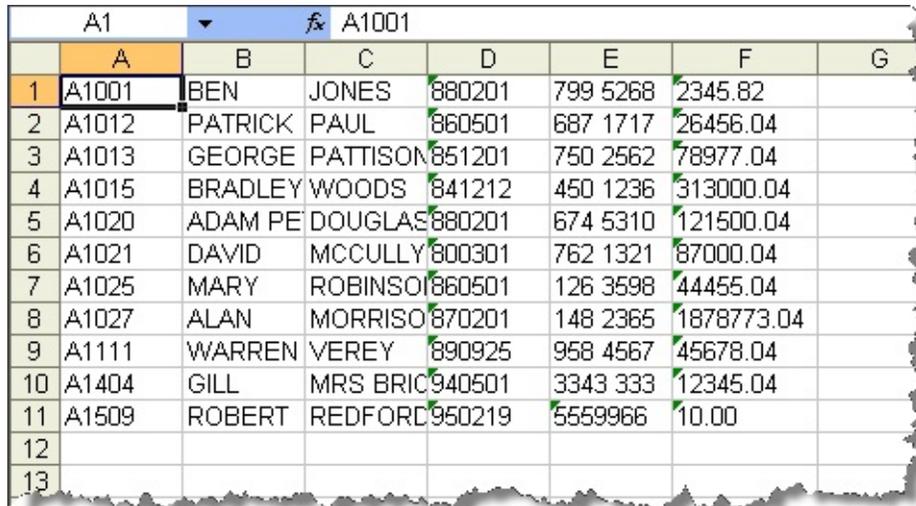
```
FUNCTION OPTIONS(*DIRECT)
BEGIN_LOOP
REQUEST FIELDS(#DEPARTMENT #SECTION)
BEGINCHECK
FILECHECK FIELD(#SECTION) USING_FILE(SECTAB)
USING_KEY(#DEPARTMENT #SECTION) MSGTXT('Department/Section not
found')
FILECHECK FIELD(#section) USING_FILE(PSLMST1)
USING_KEY(#deptment #section) MSGTXT('No employees for this
department/section')
ENDCHECK
* Call JSM function to create Excel
EXCHANGE FIELDS(#DEPARTMENT #SECTION)
CALL PROCESS(*DIRECT) FUNCTION(iiifn12)
END_LOOP
```

4. Compile your function. If you are using the configuration file jsmcltda.txt to point to the IBM I for the JSM server, you can run function iiiFN13 in Windows. Alternatively, check in and compile it, and test using a 5250 emulator.

Test your *Create Excel* application. Remember your document will be written to

the training subdirectory of the JSM Instance. If necessary, create the folder.../jsm/instance/training.

Open your document in Excel. It should look like the following:



	A1	A1001					
	A	B	C	D	E	F	G
1	A1001	BEN	JONES	880201	799 5268	2345.82	
2	A1012	PATRICK	PAUL	860501	687 1717	26456.04	
3	A1013	GEORGE	PATTISON	851201	750 2562	78977.04	
4	A1015	BRADLEY	WOODS	841212	450 1236	313000.04	
5	A1020	ADAM PE	DOUGLAS	880201	674 5310	121500.04	
6	A1021	DAVID	MCCULLY	800301	762 1321	87000.04	
7	A1025	MARY	ROBINSON	860501	126 3598	44455.04	
8	A1027	ALAN	MORRISON	870201	148 2365	1878773.04	
9	A1111	WARREN	VEREY	890925	958 4567	45678.04	
10	A1404	GILL	MRS BRICE	940501	3343 333	12345.04	
11	A1509	ROBERT	REDFORD	950219	5559966	10.00	
12							
13							

Note: If you are using the JSM Server on an IBM i server you will need to map a drive to folder /training. If you are using a local JSM Server the folder will be \JSMInstance\training.

5. If you have problems, remember you can use debug to follow your program logic. You could also start the ExcelReadService with trace and check the trace files. To do this add TRACE(*YES) to SERVICE_LOAD of the ExcelReadService.

```
#jsmxcmd := 'SERVICE_LOAD SERVICE(ExcelReadService)
TRACE(*YES)'
```

Summary

Important Observations

- The ExcelReadService provides a means to create and read Microsoft Excel documents
- You could use this to create documents to exchange between offices or with trading partners
- You could also extract data from the corporate database for analysis and presentation purposes

Tips & Techniques

- Data can be read from or written to a worksheet in a specified row and column range using a LANSAs working list or a 3GL data structure
- See *Appendix B* in the *LANSAs Integrator* guide for RDML and RDMLX examples using ExcelReadService.
- To create Excel files using the XSLX format, use the ExcelService.

What I Should Know

- How to use the ExcelReadService to write to an Excel worksheet

INT012 - Create Excel Document with Template and Formatting

Objectives

- To extend the function written in exercise INT011
- To write an Excel document based on a template
- To format the output written to the Excel document

To achieve these objectives you will complete the following:

- [Step 1. Create RDMLX function iiiFN14](#)
- [Step 2. Complete Function iiiFN14](#)
- [Step 3. Create Function iiiFN15 and test](#)
- [Step 4. Add DEFINE Command to Format Output](#)
- [Step 5. Add DEFINE Command to Format Output](#)
- [Step 6. Insert a Total Salary Formula](#)
- [Step 7. Password Protect the Document](#)
- [Summary](#)

Before You Begin

You must have Microsoft Excel or the Excel Viewer installed to complete this exercise.

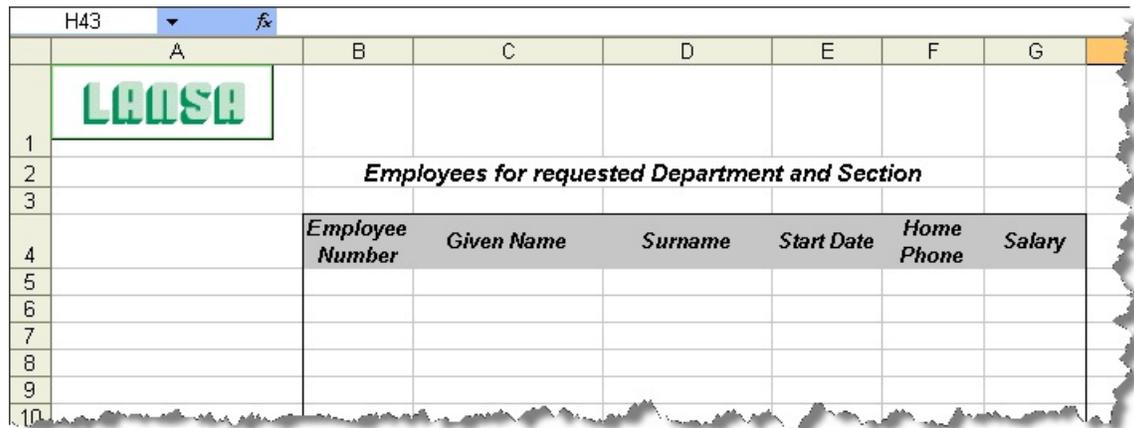
Step 1. Create RDMLX function iiiFN14

1. Select function **iiiFN12** on the *Favorites* tab. Use the context menu to *Copy* and create function **iiiFN14**, belonging to *Process IIIPRO09*.

Step 2. Complete Function iiiFN14

In this step you will use an Excel template called `Employs.xlt` which looks like the image following.

Create the Excel template (XLT) before continuing.



	A	B	C	D	E	F	G
1	LAUSA						
2	<i>Employees for requested Department and Section</i>						
3							
4	<i>Employee Number</i>	<i>Given Name</i>	<i>Surname</i>	<i>Start Date</i>	<i>Home Phone</i>	<i>Salary</i>	
5							
6							
7							
8							
9							
10							

Note:

The list of employees will need to be placed in Row 5, Column 2.

The list of employees area has a left and right border, which will need to be defined before writing the Excel document.

1. Open and review your copy of the `Employs.xlt` template.
2. To write the Excel document using the template you need to add the `TEMPLATE` keyword to the `ExcelReadService OPEN` command. Your code should look like the following:

```
#jsmxcmd := 'OPEN FILE(training/iiiemp' + #datetimec + '.XLS)
MODE(*WRITE) TEMPLATE(training/employs.xlt)'
USE BUILTIN(jsmx_command) WITH_ARGS(#jsmxhdl1 #jsmxcmd)
TO_GET(#jsmsts #jsmmsg)
EXECUTE SUBROUTINE(CHECK_STS)
```

3. You now need to write the working list of employees to Row 5, Column 2. Modify your `WRITE` command as required. Your code should look like the following:

```
* Write to Excel file
#jsmxcmd := 'WRITE R1C1(5,2)'
USE BUILTIN(jsmx_command) WITH_ARGS(#jsmxhdl1 #jsmxcmd)
TO_GET(#jsmsts #jsmmsg #employs)
```

EXECUTE SUBROUTINE(CHECK_STS)

4. Compile function iiiFN14.

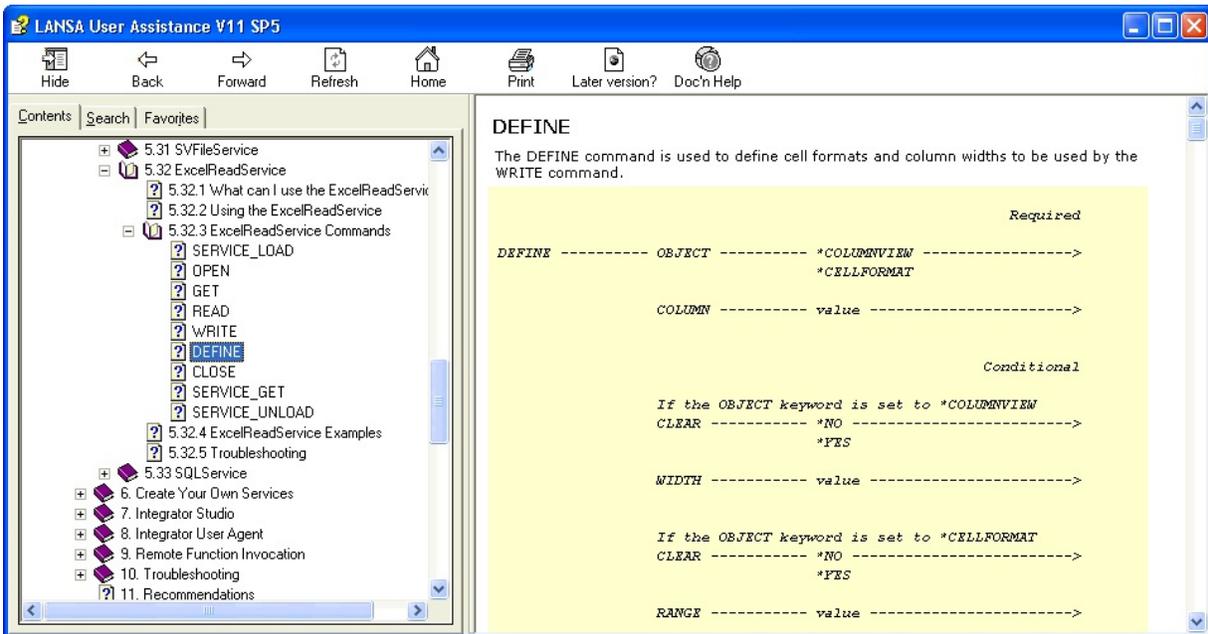
Step 3. Create Function iiiFN15 and test

1. Select function **iiiFN13** on your *Last Opened* tab and use the context menu to *Copy* it, to create **iiiFN15** belonging to process **iiiPRO09**.
2. Modify the new function to call **iiiFN14** and compile it.
3. Test the initial version of your *Create Excel with Template* application.
4. Open the document in Excel and check its contents. You should be writing the output to the correct rows and columns. Note that at the moment you are not formatting the numeric data correctly. Your Excel document should look like the following:

	A	B	C	D	E	F	G
1	LAUSA						
2	<i>Employees for requested Department and Section</i>						
3							
4		<i>Employee Number</i>	<i>Given Name</i>	<i>Surname</i>	<i>Start Date</i>	<i>Home Phone</i>	<i>Salary</i>
5		A1001	BEN	JONES	880201	799 5268	2345.82
6		A1012	PATRICK	PAUL	860501	687 1717	26456.04
7		A1013	GEORGE	PATTISON	851201	750 2562	78977.04
8		A1015	BRADLEY	WOODS	841212	450 1236	313000.04
9		A1020	ADAM PETER	DOUGLAS	880201	674 5310	121500.04
10		A1021	DAVID	MCCULLY	800301	762 1321	87000.04
11		A1025	MARY	ROBINSON	860501	126 3598	44455.04
12		A1027	ALAN	MORRISON	870201	145 7367	1279778.04

Step 4. Add DEFINE Command to Format Output

1. Review the DEFINE command of the ExcelReadService in the Integrator Guide:



Note that the DEFINE command must be used to set the cell format **before** using the WRITE command.

2. Based on the information provided in the *Integrator Guide*, consider how to format the Salary column as a number with 2 decimal places. You will usually need to use a number of DEFINE commands to format different aspects of your Excel document. In this case you will need to define OBJECT, COLUMN, RANGE, TYPE and FORMAT keywords. Your added code in iiiFN14 should look like the following:

```
#jsmxcmd := 'DEFINE OBJECT(*CELLFORMAT) COLUMN(7)
RANGE(5,38) TYPE(*NUMBER) FORMAT(*FORMAT3)
USE BUILTIN(jsmx_command) WITH_ARGS(#jsmxhdl1 #jsmxcmd)
TO_GET(#jsmsts #jsmmsg)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDL1)
```

Remember to add this code before the *ExcelReadService* **WRITE** command is processed.

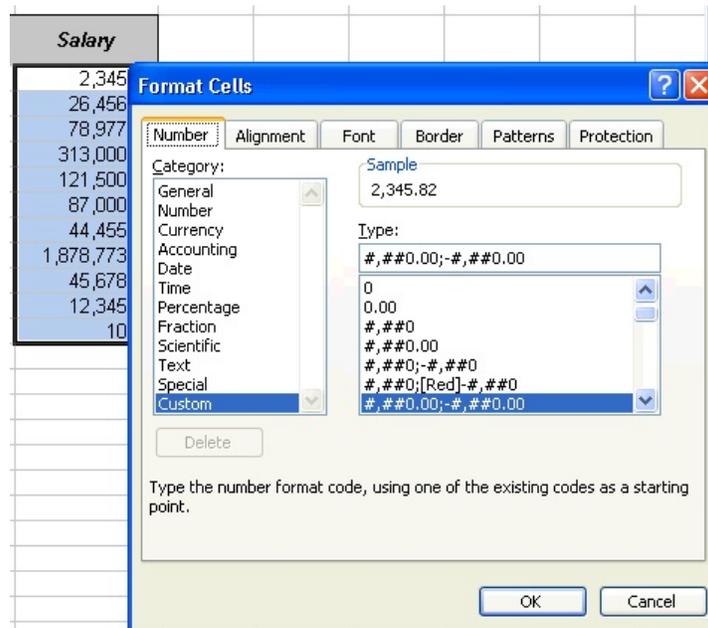
3. Recompile function iiiFN14 and retest.
4. Open the document with Excel and check your results. Your results should

look like the following:

	A	B	C	D	E	F	G
1	LAOSA						
2		<i>Employees for requested Department and Section</i>					
3							
4		Employee Number	Given Name	Surname	Start Date	Home Phone	Salary
5		A1001	BEN	JONES	880201	799 5268	2,345.82
6		A1012	PATRICK	PAUL	860501	687 1717	26,456.04
7		A1013	GEORGE	PATTISON	851201	750 2562	78,977.04
8		A1015	BRADLEY	WOODS	841212	450 1236	313,000.04
9		A1020	ADAM PETER	DOUGLAS	880201	674 5310	121,500.04
10		A1021	DAVID	MCCULLY	800301	762 1321	87,000.04
11		A1025	MARY	ROBINSON	860501	126 3598	44,455.04
12		A1027	ALAN	MORRISON	870201	148 2365	1,878,773.04
13		A1111	WARREN PETER	VEREY	890925	958 4567	45,678.04
14		A1401	SHANE	MRS BRICK	841212	3312 333	12,345.04

Note that the Salary column is now formatted correctly, but the right hand border for the Salary column has been lost.

You can check the format of the Salary column by selecting the cells and using the right mouse menu / Format Cells.



5. You can extend the existing DEFINE command to add a thin border to the right hand side of the Salary column. To do this add BORDERTYPE and BORDER keywords to the DEFINE command. Your code should look like the following:

```
#jsmxcmd := 'DEFINE OBJECT(*CELLFORMAT) COLUMN(7)
```

RANGE(5,38) TYPE(*NUMBER) FORMAT(*FORMAT3)
 BORDERSTYLE(*THIN) BORDER(*RIGHT)
 USE BUILTIN(jsmx_command) WITH_ARGS(#jsmxhdl1 #jsmxcmd)
 TO_GET(#jsmsts #jsmmsg)
 EXECUTE SUBROUTINE(CHECK_STS)

	A	B	C	D	E	F	G
1	LAOSA						
2		<i>Employees for requested Department and Section</i>					
3							
4		<i>Employee Number</i>	<i>Given Name</i>	<i>Surname</i>	<i>Start Date</i>	<i>Home Phone</i>	<i>Salary</i>
5		A1001	BEN	JONES	880201	799 5268	2,345.82
6		A1012	PATRICK	PAUL	860501	687 1717	26,456.04
7		A1013	GEORGE	PATTISON	851201	750 2562	78,977.04
8		A1015	BRADLEY	WOODS	841212	450 1236	313,000.04
9		A1020	ADAM PETER	DOUGLAS	880201	674 5310	121,500.04
10		A1021	DAVID	MCCULLY	800301	762 1321	87,000.04
11		A1025	MARY	ROBINSON	860501	126 3598	44,455.04
12		A1027	ALAN	MORRISON	870201	148 2365	1,878,773.04
13		A1028	MARY PETER	WENBY	800301	958 2357	44,455.04

Step 5. Format Start Date Column

One problem with the current solution is the lack of formatting of the Start Date column. Start Date is based on a signed numeric field, 6 digits long, In this step you will convert it to a Date type field and include this in the working list. You will also then be able to format the Start Date column in Excel as Date cells.

1. You will now be reading STARTDTER from the file, converting it and including a date work field in the working list. Change your function to include a GROUP_BY for the fields to be read and use this on the SELECT command. Your code should look like the following:

```
GROUP_BY NAME(#EMP_GRP) FIELDS(#empno #surname #givename  
#startdter #phonehme #salary)
```

```
.....
```

```
SELECT FIELDS(#EMP_GRP) FROM_FILE(pslmst1)  
WITH_KEY(#deptment #section) NBR_KEYS(*compute)
```

2. In the working list, replace field STARTDTER with STD_DATEX
3. Within the SELECT loop add logic to:

- Check that STARTDTER is a date (of YYMMDD format)
 - If so, convert STARTDTER to field STD_DATEX
 - If not, set STD_DATEX to *SQLNULL

Your code should look like the following:

```
CLR_LIST NAMED(#employs)  
SELECT FIELDS(#EMP_GRP) FROM_FILE(pslmst1)  
WITH_KEY(#deptment #section) NBR_KEYS(*compute)  
IF (#STARTDTER.IsDate( YYMMDD ))  
#Std_Datex := #STARTDTER.AsDate( YYMMDD )  
ELSE  
#Std_Datex := *SQLNULL  
ENDIF  
ADD_ENTRY TO_LIST(#employs)  
ENDSELECT
```

4. Change the DEF_LIST command for EMPLOYIS to include a counter using field LISTCOUNT. Your code should look like the following:

```
Def_List Name(#employs) Fields(#empno #givename #surname
```

```
#STD_DATEX #PHONEHME #salary) Counter(#listcount) Type(*working)
Entrys(100)
```

5. Add a DEFINE command which formats the Start Date column as a Date. Your code should look like the following:

```
* Format Start Date Column
```

```
#std_num := #listcount + 4
```

```
#jsmxcmd := 'DEFINE OBJECT(*CELLFORMAT) COLUMN(5)
```

```
RANGE(5,' + #std_num.asstring + ') TYPE(*DATE) FORMAT(*FORMAT2)'
```

```
Use Builtin(jsmx_command) With_Args(#jsmxhdle1 #jsmxcmd)
```

```
To_Get(#jsmsts #jsmmsg)
```

```
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
```

Once again, ensure the DEFINE command is before the WRITE command for the *ExcelReadService*.

6. You may have noticed when opening the document in Excel that the width of the Salary column needs adjusting because of the size of one or more salaries (e.g. 1,234,456.50). To set the width you need to use OBJECT(*COLUMNVIEW) on the DEFINE command. Your code should look like the following:

```
* Set Column width - Salary
```

```
#jsmxcmd := 'DEFINE OBJECT(*COLUMNVIEW) COLUMN(7)
```

```
WIDTH(12)'
```

```
USE BUILTIN(jsmx_command) WITH_ARGS(#jsmxhdle1 #jsmxcmd)
```

```
TO_GET(#jsmsts #jsmmsg)
```

```
EXECUTE SUBROUTINE(CHECK_STS) With_Parms(#JSMXHDLE1)
```

7. Recompile function iiiFN14 and retest. Your document should now look like the following:

A1		A	B	C	D	E	F	G
1		LAOSA						
2		Employees for requested Department and Section						
3								
4		Employee Number	Given Name	Surname	Start Date	Home Phone	Salary	
5		A1001	BEN	JONES	01-Feb-88	799 5268	2,345.82	
6		A1012	PATRICK	PAUL	01-May-86	687 1717	26,456.04	
7		A1013	GEORGE	PATTISON	01-Dec-85	750 2562	78,977.04	
8		A1015	BRADLEY	WOODS	12-Dec-84	450 1236	313,000.04	
9		A1020	ADAM PETER	DOUGLAS	01-Feb-88	674 5310	121,500.04	
10		A1021	DAVID	MCCULLY	01-Mar-80	762 1321	87,000.04	
11		A1025	MARY	ROBINSON	01-May-86	126 3598	44,455.04	
12		A1027	ALAN	MORRISON	01-Feb-87	148 2365	1,878,773.04	
13		A1111	WARREN PETER	VEREY	25-Sep-89	958 4567	45,678.04	
14		A1404	GILL	MRS BRICK	01-May-94	3343 333	12,345.04	
15		A1500	DAVID	BRICK	01-May-94	3343 333	12,345.04	

Step 6. Insert a Total Salary Formula

In this step you will use the ADD command to insert a formula to total the salary column.

1. Immediately following the SELECT/ENDSELECT loop which loads the list of employees add code to insert a total line. Your code should look like the following:

```
* Insert total line
#employs := *null
#surname := 'Salary Total:'
Add_Entry To_List(#employs)
```

2. Before the WRITE to Excel logic, add a DEFINE command to format cells in the total line. This needs to format the cell in the Start Date column, so that it is blank on the total line. Add the following code:

```
* Format total line
#std_num := #listcount + 4
#jsmxcmd := 'DEFINE OBJECT(*CELLFORMAT) COLUMN(5) RANGE('
+ #std_num.asstring + ',' + #std_num.asstring + ') TYPE(*BLANK)'
Use Builtin(jsmx_command) With_Args(#jsmxhdl1 #jsmxcmd)
To_Get(#jsmsts #jmmmsg)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
```

Review the above code.

- The RANGE parameter needs to contain the range of rows to be formatted. It must be of the form RANGE(n,n)
- The total row position needs to be calculated based on the number of lines added to the employees working list

3. Following the WRITE to Excel logic, insert the following code:

```
* Add Total Salary Formula
#std_num := (#listcount + 3)
#listcount += 4
*
#jsmxcmd := 'ADD OBJECT(*FORMULA) FORMULA("SUM(G5:G' +
#std_num.asstring + ')") R1C1(' + #listcount.asstring + ',7)'
Use Builtin(jsmx_command) With_Args(#jsmxhdl1 #jsmxcmd)
To_Get(#jsmsts #jmmmsg)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
```

Note:

- STD_NUM provides the last row number for Excel SUM formula.
 - LISTCOUNT provides the row number to insert the formula. The value of LISTCOUNT has been increased by 4, because the list is inserted into row five of the Excel document.
 - The Excel formula must be enclosed in double quotes.
 - The formula should not include the "=" symbol. This will be inserted automatically.
 - The ADD command supports the insert of formula, image, setting and hyperlink. See the *LANSA Integrator Guide* for further details.
4. Recompile your function and test it to check that your formula has been correctly inserted.

Step 7. Password Protect the Document

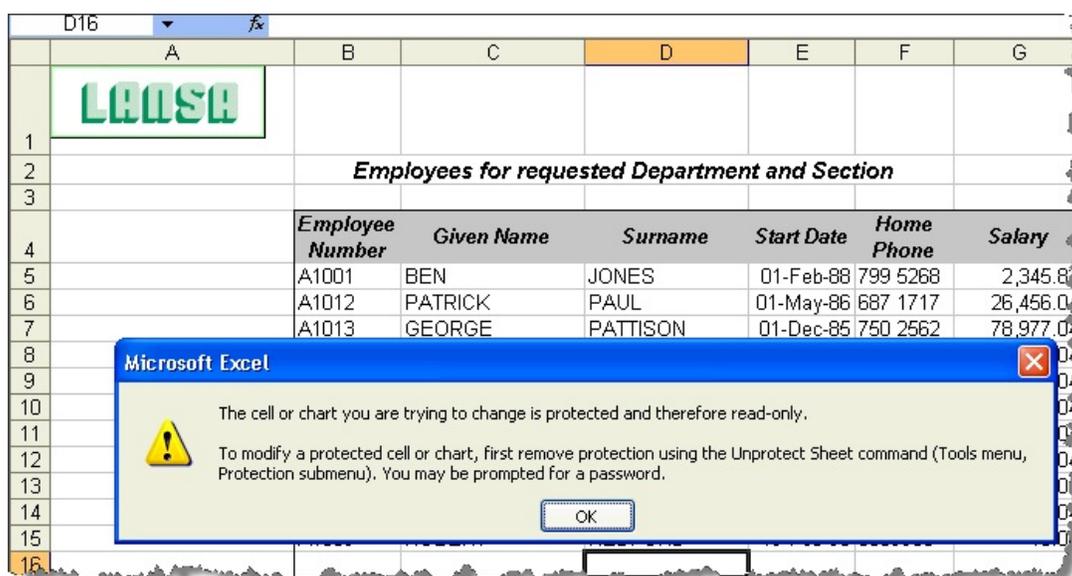
In this step you will add PROTECT and PASSWORD parameters to the WRITE command.

1. Review the ExcelReadService WRITE command in the Integrator Guide.
2. Add PROTECT and PASSWORD parameters to your WRITE command. Your code should look like the following:

* Write to Excel file

```
#jsmxcmd := 'WRITE R1C1(5,2) PASSWORD(LANSA) PROTECT(*YES)'
```

3. Compile function iiiFN14 and retest it. If you try to change the Excel document you will now see the following:



You have completed this exercise. If you have time available try to make other format changes to your Excel Document.

Summary

Important Observations

- Use the DEFINE command before the WRITE command to format cells and columns.
- You will usually need to use a number of DEFINE commands.

Tips & Techniques

- The WRITE command has PROTECT and PASSWORD parameters which enable the worksheet to be protected.

What I should now know

- How to format and protect a worksheet.

INT013 - Create PDF Output using PDFDocumentService

Objectives

- To create an application which produces a PDF document containing employee details and skills.
- To demonstrate how to use the PDFDocumentService.
- To understand how to define the PDF document layout, by defining an XML template document.

To achieve the objectives you will complete the following:

- [Step 1. Plan the PDF Page Content](#)
- [Step 2. Create the PDF XML Template](#)
- [Step 3. Create an RDMLX Function to Produce a Single Page PDF](#)
- [Step 4. Create Function iiiFN17 - Create PDF Output for Employees](#)
- [Step 5. Create Function iiiFN18 – Test PDF Output for Employees](#)
- [Summary](#)

Concepts

- The *PDFDocumentService* enables complex and dynamic PDF output to be produced.
- The format of the output PDF document is defined by an XML template document. This XML template includes <content></content> tags which are used to define the content of the various areas of a page. This structure is completely flexible, but will typically include content areas such as header, details, lists and footer.
- Your RDML/RDMLX program creates a PDF document using the XML template and "adds content" into the content areas defined by the XML template
- Content can be added in any sequence
- The page area is defined as a grid. A4 landscape for example, is defined as having an area of 590 x 840 pts.
- The bottom left hand position is 0,0
- All output to the page is defined as having a position on an X1, Y1 axis. These values define the top left hand corner for this element. Y is the vertical axis.

- In the *LANSA Integrator* guide, refer to [XML Content](#) in the *PDFDocumentService* for details of all the elements which can be defined in the XML template.
- The *PDFDocumentService* includes a [sample XML template](#) and the [RDML function](#) to output a PDF document using it.

Before You Begin

If you plan to run this exercise on your IBM i server:

- Create the following folder, if it does not already exist:

/LANSA_<pgmlib>/jsm/instance/training

Where <pgmlib> is your LANSAs program library, e.g. dcxpgmlib.

- Copy these files to the . . /training folder
- lanlogo.gif
- ph0070.tif

These files are included in a zip file which can be download from:

<http://www.lansa.com/support/docs/index.htm>

See LANSAs Integrator / Extra Tutorial Files.

Step 1. Plan the PDF Page Content

ACME Manufacturing Company, Chicago



BLOGGS, FRED JOHN ALAN
70 MAIN STREET
NEWTOWN NSW
AUSTRALIA
2220

Business Tel: 654 6475 X432
Home Tel: 344-2234454545
Department: INFORMATION SERVICES
Section: ACCOUNTING

POWERED BY
LALSA

SKILL	GRADE	DATE ACQUIRED	COMMENT
Administratn Part 1	Distinction	25/03/1998	Met requirement
Administratn Part 2	Pass	01/05/1998	G
Communications Degre	Distinction	04/05/1998	
Computer Science Deg	Pass	05/05/1998	
History Degree	Fail	02/05/1998	EXCELLENT
History Degree	Pass	05/05/1998	

Each page will contain:

- A fixed heading and logo image (lanlogo.gif). The application will provide the heading text in field STD_TEXTL
- A hard code employee image (ph0070.tif). In a real application this would be a variable supplied by the application program
- Employee details
- Employee skills
- A footer contain report date and page number (not shown in the above image)

The Date Acquired will be output as a text field (STD_TEXT) with the date formatted as DD/MM/YYYY

The report date will be a text field STD_NAME with the current date in DD/MM/YYYY

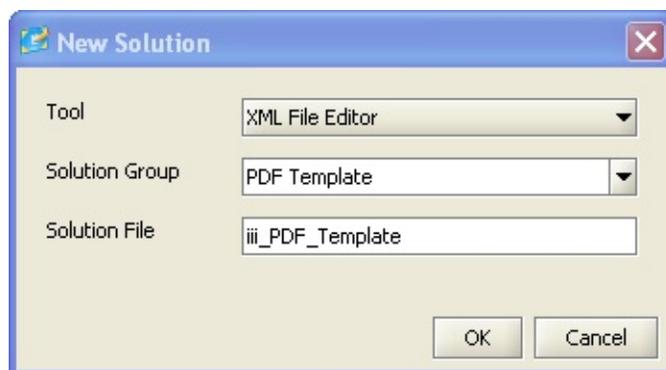
Page number will be output in field REP1PAGE

Initially a simple RDMLX function will be developed, to output a single page for a fixed employee number.

A second version of the report function will be called by a form or an RDML function, passing EMPNO, DEPARTMENT, SURNAME and the type of report required (STD_FLAG). This report function will produce output based on reading the Employee file (PSLMST) based on EMPNO, DEPARTMENT or SURNAME.

Step 2. Create the PDF XML Template

1. Open Integrator Studio.
2. Open the **iii Training** project. Create a new *Solution Group / PDF Template*.
3. In the *PDF Template* group, create a *New Solution*, using the *XML File Editor*. Create a solution file **iii_PDF_Template.xml**, where **iii**=your initials.



4. Copy and paste the XML code from [INT013 - Appendix A](#), to replace the code in your new XML file. Save your changes.

The supplied XML template is incomplete. You will complete the template by adding elements for employee details and the skills list.

5. Review the contents of the template.
 - The PDF document template is defined within `<document></document>` tags.
 - The page size is defined as A4. Other attributes are defined at document level. The orientation is defined as landscape.
 - The template contains a number of `<content></content>` tags. The application program ADDs CONTENT() to these areas using the PDFDocumentService.
 - Styles are defined at the end of the XML document within `<style />` tags.
 - Elements within the template refer to these styles. For example:

```
<text x1="66" y1="70" width="150" height="15" style="footer" border="" align="left" leading="" value="{STD_NAME}"/>
```
6. Locate this line: `<content name="HEADER">` in the template:

```

<content name="HEADER">
<!--
<!-- Applicant Details-->

<text x1="66" y1="540" width="350" height="25" style="title" border="" align="left" leading="" value="{STD_TEXTL}"/>
<image x1="66" y1="490" file="..\x_win95\x_lansa\x_dem\execute\ph0070.tif" scale="10" />
<text x1="240" y1="490" width="150" height="15" style="normal" border="" align="left" leading="" value="{FULLNAME}"/>

<text x1="530" y1="490" width="250" height="15" style="label" border="1" align="left" leading="" value="Business Tel:"/>
<text x1="530" y1="480" width="149" height="15" style="label" border="" align="left" leading="" value="Home Tel:"/>

```

The x1 and y1 values in each element define the top left position for that element on the page, measured from the bottom left corner of the page. Y1 is the vertical component.

- a. Add four text elements below FULLNAME with the same left hand location (x1="240") and with y1= a value 10pts below the existing FULLNAME line.
- a. Add entries for fields ADDRESS1, ADDRESS2, ADDRESS3 and POSTCODE. To do this, copy the FULLNAME line and then modify it.

Your new code should look like the following:

```

<text x1="240" y1="480" width="150" height="15" style="normal" border=""
align="left" leading="" value="{ADDRESS1}"/>
<text x1="240" y1="470" width="150" height="15" style="normal" border=""
align="left" leading="" value="{ADDRESS2}"/>
<text x1="240" y1="460" width="150" height="15" style="normal" border=""
align="left" leading="" value="{ADDRESS3}"/>
<text x1="240" y1="450" width="150" height="15" style="normal" border=""
align="left" leading="" value="{POSTCODE}"/>

```

7. Save your changes.
8. Locate the following position in the template:

```

<content name="SKILLSTBL">
<!-- TABLE using working list -->
<!-- ***** -->
<table x1="66" y1="390" width="700" height="500" style="normal" alternate="false" title-show="" border="false" cell-border="false">
<column field="SKILDESC" width-percentage="20" vertical-align="center" horizontal-align="left" title-horizontal-align="left" title-border="false">
<column field="GRADEDES" width-percentage="20" vertical-align="center" horizontal-align="left" title-horizontal-align="left" title-border="false">
</table>
</content>

```

This defines a table that will be populated from a working list of employee skills.

9. To complete the skills table:
 - a. Add entries for STS_TEXT (Date Acquired) and COMMENT. Copy and paste from the entry for GRADEDES to create two new lines and then

modify their content.

b. Change width-percentage to 30 and change title as well as the field value.

Your completed new code should look like the following:

```
<column field="STD_TEXT" width-percentage="30" vertical-align="center"
horizontal-align="left" title-horizontal-align="left" title-border="false" title-
style="table" title="DATE ACQUIRED" />
```

```
<column field="COMMENT" width-percentage="30" vertical-align="center"
horizontal-align="left" title-horizontal-align="left" title-border="false" title-
style="table" title="COMMENT" />
```

10. Save your changes.

11. Review the content for FOOTER1. Note that this will require STD_NAME (current date) and REP1PAGE field values.

12. Note that the FOOTER1 content includes:

```
<page orientation="landscape"/>
```

When the FOOTER1 content is added, a new page will be added, with orientation="landscape".

13. If you will be running this exercise on your IBM i server, locate the following line:

```
<image x1="710" y1="540" file="..\..\webserver\images\lanlogo.gif"
scale="100" />
```

Replace it with:

```
<image x1="710" y1="540" file="training/lanlogo.gif" scale="100" />
```

Locate the line:

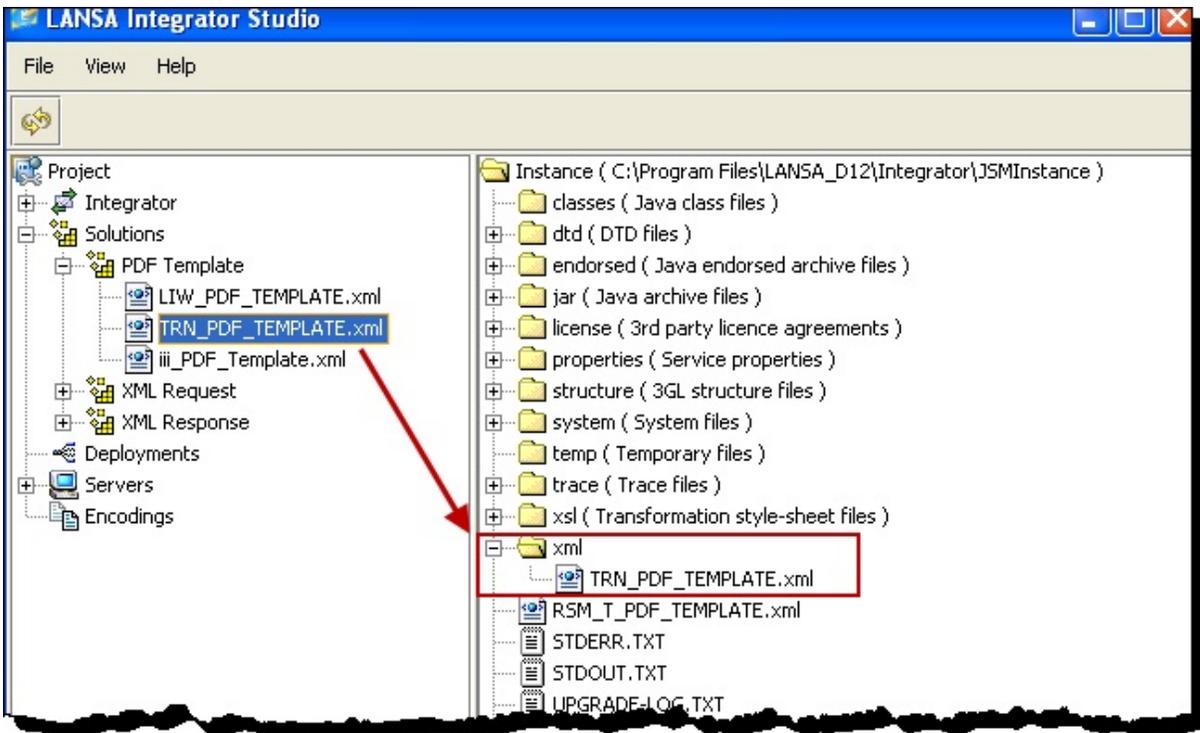
```
<image x1="66" y1="490"
file="..\..\x_win95\x_lansa\x_dem\execute\ph0070.tif" scale="10" />
```

Replace it with:

```
<image x1="66" y1="490" file="training/ph0070.tif" scale="10" />
```

14. Save your changes and exit the XML Editor

15. With your *iii Training* project open in Integrator Studio, open the server that you are using. This could be localhost, if you are using Integrator installed and licensed on your PC, or it could be the IBM i server used for training.



16. Create an XML folder (directory) in the server instance. Note that the folder may already exist. Drag and drop (copy) your iii_PDF_Template.xml to the server.

17. You now have a PDF XML Template ready to use, and you can write the RDMLX function to produce the PDF document.

Step 3. Create an RDMLX Function to Produce a Single Page PDF

In this step you will create an RDMLX function, based on the RDMLX source in [INT013 - Appendix B](#). You will complete this function to produce a PDF document for a single employee.

1. Create a new process **iiiPRO10 – PDF Examples**.
2. Create a new RDMLX enabled function **iiiFN16 – Create PDF for an Employee**, belonging to process **iiiPRO10**, without using a template.
3. Replace its code with the RDMLX source from INT013. Appendix B.
4. Save your function and review its content. Note that at the beginning of the program a number of field values are hard coded, such as EMPNO and STD_TEXTL. This function will process a single fixed employee number.
5. Find the following comment:

* Create PDF Document

Following this comment, add code to use the PDFDocumentService which has already been loaded. Use the CREATE command to create a PDF document based on the template iii_PDF_Template.xml.

6. In your source code, copy and paste the three statements from the * Load Service code and modify the CHANGE Field(#JSMXCMD) command to define the CREATE command:

Keyword	Value
DOCUMENT	iii_EMPREP.pdf
CONTENT	xml/iii_PDF_Template.xml

In a real application you would want to write the output to a specific folder, perhaps organized by user or customer.

Your completed code should look like the following.

```

* Create PDF Document
Change Field(#JSMXCMD) To('CREATE DOCUMENT(iii_EMPREP.pdf)
CONTENT(xml/iii_PDF_Template.xml)')
Use Builtin(JSMX_COMMAND) With_Args(#jsmxhdl1 #JSMXCMD)
To_Get(#JSMSTS #JSMMSG)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)

```

7. Save your changes.

8. Locate this comment:

```
* If found add content
```

The code immediately preceding this comment has created a single entry working list (#HEADER) containing the required employee detail fields.

9. Following this comment, add logic to add content HEADER containing employee details.

Add this code and then complete the * Add content HEADER logic:

```

If_Status Is(*OKAY)
* Add content HEADER
Else
Message Msgtxt('Employee A0090 not found')
Endif

```

10. Following the * Add content HEADER comment line, define an ASSIGN command to set up the JSMXCMD field to ADD based on:

Keyword	Value
CONTENT	HEADER

11. Define the USE the JSMX_COMMAND based on:

Keyword	Value
WITH_ARGS	#JSMXHDLE1
	#JSMXCMD

	#HEADER
TO_GET	#JSMSTS
	#JSMMSG

Note: You are passing the working list #HEADER as the third argument.

12. Finally add an EXECUTE command to perform subroutine to check JSM status.

Your completed code should look like the following:

```
If_Status Is(*OKAY)
#JSMXCMD := 'ADD CONTENT(HEADER)'
Use Builtin(JSMx_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD
#HEADER) To_Get(#JSMSTS #JSMMSG)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
Else
Message Msgtxt('Employee A0090 not found')
Endif
```

13. Save your changes

14. Locate the following comment:

* If found add employee skills list

15. Add the following logic and complete your JSM logic within it:

```
If_Status Is_Not(*error)
* Add SKILLTBL content
```

```
Else
Message Msgtxt('Employee skills not found')
Endif
```

16. Following the * Add SKILTBL content comment, add a CHANGE command to change the JSMXCMD field to ADD based on:

Keyword	Value

CONTENT	SKILLTBL
---------	----------

17. Add a USE command to perform the JSMX_COMMAND BIF, based on:

Keyword	Value
WITH_ARG	#JSMXHDLE1
	#JSMXCMD
TO_GET	#JSMSTS
	#JSMMSG
	#SKILLTBL

18. Add an EXECUTE command to perform the subroutine to check JSM status.

Your completed code should look like the following:

```

If_Status Is_Not(*error)
* add SKILLTBL content
Change Field(#JSMXCMD) To('ADD CONTENT(SKILLTBL)')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
To_Get(#JSMSTS #JSMMSG #SKILLTBL)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
Else
Message Msgtxt('Employee skills not found')
Endif

```

19. Compile your function. If you are running Integrator on an IBM i server, check in and compile your function on the server.

20. Execute your function iiiFN16 – Create a PDF for a Single Employee.

Find the output PDF document in the root of your JSM instance. If you are running on the IBM i server, this will be a path such as:

/LANSA_<pgmlib>/jsm/instance

where <pgmlib> is the LANSAs library name.

The PDF document should look like the following:

ACME Manufacturing Company, Chicago			
	BLOGGS, FRED JOHN ALANI 70 MAIN STREET NEWTOWN NSW AUSTRALIA 2220	Business Tel: N/A Home Tel: 344-2234454545 Department: FLEET ADMINISTRATION Section: ACCOUNTING	
SKILL	GRADE	DATE ACQUIRED	COMMENT
Administratn Part 1	Distinction	25/03/1998	Met requirement
Administratn Part 2	Pass	01/05/1998	G
Communications Degre	Distinction	04/05/1998	
Computer Science Deg	Pass	05/05/1998	
History Degree	Fail	02/05/1998	EXCELLENT
Car			

Step 4. Create Function **iiiFN17** - Create PDF Output for Employees

In this step you will copy function **iiiFN16** and extend it to output a PDF document for a single or multiple employees, selected by employee number, department or surname.

The new RDMLX function **iiiFN17**, will be called by an RDML function **iiiFN18 – Test Employee PDF Output**, which will input STD_FLAG, EMPNO, DEPARTMENT and SURNAME, then EXCHANGE these values and call function **iiiFN17**.

A practical consideration is that the interactive function **iiiFN18**, will validate EMPNO against the file PSLMST. Function **iiiFN17**, when called passing EMPNO, the employee number will always be valid and **iiiFN17** will therefore always output a PDF document and the document should be closed.

When function **iiiFN17** is called passing DEPARTMENT or SURNAME to produce a one or more page PDF document, there may be no output. Closing the PDF document conditionally will avoid writing an empty pdf document.

1. Copy function **iiiFN16** to create an RDMLX enabled function **iiiFN17** in process **iiiPRO10**.
2. Delete these two lines from the top of the function

```
#empno := A0090
```

```
#rep1page := 1
```

3. Locate the * Add Footers comment line. Create a FOOTER subroutine and move add footers logic into it. Your code should look like the following

```
Subroutine Name(FOOTER)
```

```
Clr_list #FOOTER1
```

```
Add_entry #FOOTER1
```

```
#jsmxcmd := 'ADD CONTENT(FOOTER1)'
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDL1 #JSMXCMD  
#FOOTER1) To_Get(#JSMSTS #JSMMSG)
```

```
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDL1)
```

```
Endroutine
```

4. Add an EXECUTE subroutine to replace the code moved into the subroutine. Your code should look like the following:

* Add Footers

EXECUTE Subroutine(FOOTER)

5. Locate the * Close document comment. Create a CLOSE subroutine, and move the code shown into it:

Subroutine Name(CLOSE)

#jsmxcmd := CLOSE

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)

To_Get(#JSMSTS #JSMMSG)

Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)

#rep1page -= 1

Message Msgtxt('Employees PDF Report produced with ' +

#rep1page.asstring + ' pages')

* Unload PDF Service

#jsmxcmd := SERVICE_UNLOAD

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)

To_Get(#JSMSTS #JSMMSG)

Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)

* Close the JSM

Use Builtin(JSMX_CLOSE) With_Args(#jsmxdhle1) To_Get(#JSMSTS

#JSMMSG)

Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)

Endroutine

6. Add an Execute subroutine to replace the code moved into the subroutine. Your code should look like the following:

* Close document

EXECUTE Subroutine(CLOSE)

7. Function iiiFN17 will receive fields EMPNO, DEPARTMENT, SURNAME and STD_FLAG. STD_FLAG will contain the search request type, E=Employee Number, D=Department Code, S=Surname search.

8. Add a case loop for STD_FLAG around your existing single employee logic. The new code is shown in **red**. For example:

Case Of_Field(#STD_FLAG)

*** Search for single employee number**

When (= E)

* Add Logo

```
Change Field(#JSMXCMD) To('ADD CONTENT(LOGO) ')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
To_Get(#JSMSTS #JSMMSG)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
```

.....

.....

* Add Footer

Execute Subroutine(FOOTER1)

* Close the PDF Document and the PDF Service

Execute Subroutine(CLOSE)

Endcase

.

9. At the top of the function, define a work field OUTSTS based on field IO\$STS. This field will be used to control whether the PDF document should be closed, because output was produced.
10. Your existing CASE loop handles a single employee only. The CASE loop should now be extended by adding the next WHEN condition below EXECUTE FOOTER for the single employee. Copy the code to handle search by surname and search by department which is supplied in [INT013 - Appendix C](#).
11. Save your changes.
12. Review the logic just added, for STD_FLAG = S and STD_FLAG = D:
 - Page number, REP1PAGE is incremented after output for each employee.
 - OUTSTS is used to control the CLOSE of the PDF document, to avoid writing an empty PDF document. OUTSTS is set to OK when output has been produced for one or more employees. An alternative if no employees are output, could be to write a page containing the text "No employee details found".
 - The PDF XML template contains `<page orientation="landscape"/>` in the FOOTER1 content, which starts a new page for the next employee output.
13. Compile function iiiFN17 and correct errors if necessary.

14. If running functions on the IBM i server, check in and compile on the server.

Step 5. Create Function iiiFN18 – Test PDF Output for Employees

1. Create an RDML function, **iiiFN18 – Test PDF Output for Employees** belonging to process **iiiPRO10**.

2. Replace its code with the following:

```
FUNCTION OPTIONS(*DIRECT)
OVERRIDE FIELD(#STD_FLAG) LABEL('Report Type')
GROUP_BY NAME(#SCREEN) FIELDS((#STD_FLAG *L004 *P002)
(#EMPNO *L008 *P002) (#SURNAME *L009 *P002) (#DEPARTMENT
*L010 *P002) (#OPT01 *L006 *P021 *OUTPUT *NOID) (#OPT02 *L004
*P021 *OUTPUT *NOID) (#OPT03 *L005 *P021 *OUTPUT *NOID))
DEFINE FIELD(#OPT01) REFFLD(#std_descs) DEFAULT("E =
Employee")
DEFINE FIELD(#OPT02) REFFLD(#std_descs) DEFAULT("S = Surname")
DEFINE FIELD(#OPT03) REFFLD(#std_descs) DEFAULT("D =
Department")
BEGIN_LOOP
REQUEST FIELDS(#SCREEN) IDENTIFY(*LABEL)
BEGINCHECK
VALUECHECK FIELD(#STD_FLAG) WITH_LIST('S' 'D' 'E')
MSGTXT('Report Type must be D, E or S')
CASE OF_FIELD(#STD_FLAG)
WHEN VALUE_IS('= E')
FILECHECK FIELD(#EMPNO) USING_FILE(pslmst) MSGTXT('Employee
not found')
WHEN VALUE_IS('= D')
FILECHECK FIELD(#DEPARTMENT) USING_FILE(deptab)
MSGTXT('Department not found')
WHEN VALUE_IS('= S')
IF COND(#SURNAME = *blanks')
SET_ERROR FOR_FIELD(#SURNAME)
MESSAGE MSGTXT('Surname may not be blank')
ENDIF
ENDCASE
ENDCHECK
EXCHANGE FIELDS(#EMPNO #DEPARTMENT #SURNAME #STD_FLAG)
CALL PROCESS(*DIRECT) FUNCTION(IIIFN17) EXIT_USED(*NEXT)
MENU_USED(*NEXT)
```

END_LOOP

3. Change the called function name using your initials.
4. If running functions on the IBM i server, check in and compile on the server.
5. Test function iiiFN17 by running function iiiFN18.
6. Check the PDF document produced when selecting by employee number, by surname and by department.

Summary

Important Observations

- The PDFDocumentService enables complex and dynamic PDF documents to be produced.
- The format of the PDF document is defined in an XML template
- Your RDML produces the PDF based on the structure defined by the XML
- Your RDML "adds content" into the PDF document
- The page is defined as a grid in using units of points. An A4 landscape page is 590 x 840 points for example
- The bottom left corner is location 0,0

Tips & Techniques

- Time spent planning your document layout in some detail, will speed up the overall task

What I Should Know

- How to define the content of the PDF document using an XML template
- How to create PDF document using the PDFDocumentService.

INT013 - Appendix A

Source for PDF XML Template (iii_PDF_Template.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  List Sample PDF XML document
-->
<document page-size="A4"

  orientation = "landscape"
  background="white"
  border="false"
  author="Acme Corporation"
  title="Document Title"
  subject="Document Subject"
  keywords="word1, word2"
  creator=""
  hide-toolbar=""
  hide-menubar=""
  hide-windowui=""
  allow-printing="true"
  allow-copy="false"
  allow-modify-contents="false"
  allow-modify-annotations="false"
  allow-fillin="false"
  allow-screenreader="false"
  allow-assembly="false"
  allow-degraded-printing="false">

  <content name="LOGO">
    <image x1="710" y1="540" file="..\..\webserver\images\lanlogo.gif"
    scale="100" />
  </content>

  <content name="HEADER">
  <!--

-->
```

<!-- Applicant Details-->

<text x1="66" y1="540" width="350" height="25" style="title" border="" align="left" leading="" value="{STD_TEXTL}"/>

<image x1="66" y1="490"

file="..\..\x_win95\x_lansa\x_dem\execute\ph0070.tif" scale="10" />

<text x1="240" y1="490" width="150" height="15" style="normal" border="" align="left" leading="" value="{FULLNAME}"/>

<text x1="530" y1="490" width="250" height="15" style="label" border="1" align="left" leading="" value="Business Tel:"/>

<text x1="530" y1="480" width="149" height="15" style="label" border="" align="left" leading="" value="Home Tel:"/>

<text x1="530" y1="470" width="149" height="15" style="label" border="" align="left" leading="" value="Department:"/>

<text x1="530" y1="460" width="149" height="15" style="label" border="" align="left" leading="" value="Section:"/>

<text x1="600" y1="490" width="250" height="15" style="normal" border="" align="left" leading="" value="{PHONEBUS}"/>

<text x1="600" y1="480" width="250" height="15" style="normal" border="" align="left" leading="" value="{PHONEHME}"/>

<text x1="600" y1="470" width="250" height="15" style="normal" border="" align="left" leading="" value="{DEPTDESC}"/>

<text x1="600" y1="460" width="250" height="15" style="normal" border="" align="left" leading="" value="{SECDESC}"/>

</content>

<content name="SKILLTBL">

<!-- Table using working list -->

<!-- ***** -->

<table x1="66" y1="390" width="700" height="500" style="normal" alternate="false" title-show="" border="false" cell-border="false">

<column field="SKILDESC" width-percentage="20" vertical-align="center" horizontal-align="left" title-horizontal-align="left" title-border="false" title-style="table" title="SKILL" />

<column field="GRADEDES" width-percentage="20" vertical-align="center" horizontal-align="left" title-horizontal-align="left" title-

```
border="false" title-style="table" title="GRADE" />
```

```
</table>
```

```
</content>
```

```
<content name="FOOTER1">
```

```
<text x1="66" y1="70" width="150" height="15" style="footer" border=""  
align="left" leading="" value="{STD_NAME}"/>
```

```
<text x1="670" y1="70" width="50" height="15" style="footer" border=""  
align="right" leading="" value="Page:"/>
```

```
<text x1="710" y1="70" width="50" height="15" style="footer" border=""  
align="right" leading="" value="{REP1PAGE}"/>
```

```
<page orientation="landscape"/>
```

```
</content>
```

```
<!--
```

```
    User styles
```

```
-->
```

```
<style name="normal" file="Din.ttf" size="8" color="black"  
encoding="Cp1252" embedded="true"/>
```

```
<style name="table" file="Din.ttf" size="9" color="black" style="bold"  
encoding="Cp1252" embedded="true"/>
```

```
<style name="footer" file="Din.ttf" size="7" color="black" style="normal"  
encoding="Cp1252" embedded="true"/>
```

```
<style name="footerb" file="Din.ttf" size="7" color="black" style="bold"  
encoding="Cp1252" embedded="true"/>
```

```
<style name="footeru" file="Din.ttf" size="7" color="black"  
style="underline" encoding="Cp1252" embedded="true"/>
```

```
<style name="small" file="Din.ttf" size="7" color="black" style="normal"  
encoding="Cp1252" embedded="true"/>
```

```
<style name="title" file="Din.ttf" size="12" color="black" style="bold"  
encoding="Cp1252" embedded="true"/>
```

```
<style name="label" file="Din.ttf" size="8" color="black" style="italic"  
encoding="Cp1252" embedded="true"/>
```

```
</document>
```

INT013 - Appendix B

RDMLX Source for Function iiiFN16 – Create PDF for a single Employee

Function Options(*DIRECT)

* Beginning of RDML commands *****

#empno := A0090

#std_textl := 'ACME Manufacturing Company, Chicago'

#std_name := #datex.asdisplayString(DDsMMsCCYY)

#rep1page := 1

Def_List Name(#footer1) Fields(#std_name #rep1page) Type(*working)

Define Field(#ID) Type(*CHAR) Length(5)

Define Field(#FNAME) Type(*CHAR) Length(20)

Define Field(#SNAME) Type(*CHAR) Length(30)

Def_List Name(#SKILLTBL) Fields(#SKILDESC #GRADEDES

#STD_TEXT #COMMENT) Type(*WORKING) Entrys(*max)

Def_List Name(#header) Fields(#std_textl #fullname #address1 #address2

#address3 #postcode #phonebus #phonehme #deptdesc #secdesc)

Type(*working) Entrys(*max)

* Open JSM

Use Builtin(JSMX_OPEN) To_Get(#JSMSTS #JSMMSG #jsmxhdle1)

Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDL1)

* Load Service

#jsmxcmd := ('SERVICE_LOAD SERVICE(PDFDOCUMENTSERVICE)')

Use Builtin(JSMX_COMMAND) With_Args(#jsmxhdle1 #jsmxcmd)

To_Get(#JSMSTS #JSMMSG)

Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDL1)

* Create PDF Document

* Add Logo

Change Field(#JSMXCMD) To('ADD CONTENT(LOGO)')

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDL1 #JSMXCMD)

To_Get(#JSMSTS #JSMMSG)

Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDL1)

* Add Header Employee Info

Clr_List Named(#header)

Fetch Fields(*all) From_File(pslmst) With_Key(#empno) Val_Error(*next)

#fullname := #surname + ', ' + #givenname

Fetch Fields(#deptdesc) From_File(deptab) With_Key(#deptment)

Fetch Fields(#secdesc) From_File(sectab) With_Key(#deptment #section)

Add_Entry To_List(#header)

* If found add content

* Get employee skills

* =====

Clr_List Named(#skilltbl)

Group_By Name(#skills) Fields(#SKILCODE #COMMENT #DATEACQ
#SKILDESC #grade)

Select Fields(#skills) From_File(pslskl) With_Key(#empno)

Nbr_Keys(*compute) Val_Error(*next)

Fetch Fields(#skills) From_File(skltab) With_Key(#skilcode)

Val_Error(*next)

#std_text := #dateacq.asdate(DDMMYY).asdisplayString(DDsMMsCCYY)

Execute Subroutine(grade)

Add_Entry To_List(#skilltbl)

Endselect

* If found add employee skills list

* Add Footers

clr_list #FOOTER1

Add_entry #FOOTER1

#jsmxcmd := 'ADD CONTENT(FOOTER1)'

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDL1 #JSMXCMD
#FOOTER1) To_Get(#JSMSTS #JSMMSG)

Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDL1)

* Close document

#jsmxcmd := CLOSE

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
To_Get(#JSMSTS #JSMMSG)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
```

* Unload PDF Service

```
#jsmxcmd := SERVICE_UNLOAD
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
To_Get(#JSMSTS #JSMMSG)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
```

* Close the JSM

```
Use Builtin(JSMX_CLOSE) With_Args(#jsmxhdl1) To_Get(#JSMSTS
#JSMMSG)
```

```
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
```

```
If (#jsmsts = OK)
```

```
Message Msgtxt('Document iiiEMPREP.pdf produced')
```

```
Endif*
```

* Check the status of the JSM command issued

*

```
Subroutine Name(CHECK_STS) Parms(#W_HDLE)
```

*

```
Define Field(#MSGDTA) Type(*CHAR) Length(132)
```

```
Define Field(#W_HDLE) Type(*CHAR) Length(4)
```

*

```
If Cond('#JSMSTS *NE OK')
```

*

```
#MSGDTA := 'Error Status Code: ' + #JSMSTS
```

```
Message Msgid(DCM9899) Msgf(DC@M01) Msgdta(#MSGDTA)
```

```
#MSGDTA := 'Error Message: ' + #JSMMSG
```

```
Message Msgid(DCM9899) Msgf(DC@M01) Msgdta(#MSGDTA)
```

```
Endif
```

*

```
Def_List Name(#WL_MSGS) Fields(#JSMSTS #JSMMSG)
```

```
Type(*WORKING) Entries(*max)
```

```
Add_Entry To_List(#WL_MSGS)
```

```
Endroutine
```

```
Subroutine Name(grade)
```

```
Case (#grade)
```

```
When (= P)
```

```
#gradedes := 'Pass'  
When (= F)  
#gradedes := 'Fail'  
When (= M)  
#gradedes := 'Merit'  
When (= D)  
#gradedes := 'Distinction'  
Endcase
```

```
Endroutine
```

```
* End of RDML commands *****
```

INT013 - Appendix C

RDMLX Source code to complete iiiFN17 – Create PDF Output for Employees

```
* Produce PDF document for employees selected by surname
When (= S)
#outsts := *blanks
#rep1page := 1
Select Fields(*all) From_File(pslmst2) With_Key(#surname)
Nbr_Keys(*compute) Generic(*yes)
* Add Logo
Change Field(#JSMXCMD) To('ADD CONTENT(LOGO) ')
Use Builtin(JSMx_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)
To_Get(#JSMSTS #JSMMSG)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
* Clear employee header list
Clr_List Named(#header)

#fullname := #surname + ', ' + #givenname
Fetch Fields(#deptdesc) From_File(deptab) With_Key(#deptment)
Fetch Fields(#secdesc) From_File(sectab) With_Key(#deptment #section)
Add_Entry To_List(#header)
If_Status Is(*OKAY)
Change Field(#JSMXCMD) To('ADD CONTENT(HEADER) ')
Use Builtin(JSMx_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD
#HEADER) To_Get(#JSMSTS #JSMMSG)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
#outsts := OK
Else
Message Msgtxt('Employees not found for surname: ' + #surname)
Leave
Endif
* Add Employee Skills List
* Get employee skills
* =====
Clr_List Named(#skilltbl)
Select Fields(#skills) From_File(pslskl) With_Key(#empno)
Nbr_Keys(*compute) Generic(*yes) Val_Error(*next)
```

```
Fetch Fields(#skills) From_File(skltab) With_Key(#skilcode)
Val_Error(*next)
#std_text := #dateacq.asdate( DDMMYY ).asdisplayString( DDsMMsCCYY )
Execute Subroutine(grade)
Add_Entry To_List(#skilltbl)
Endselect
```

```
If_Status Is_Not(*error)
Change Field(#JSMXCMD) To('ADD CONTENT(SKILLTBL)
SERVICE_LIST(SKILDESC,GRADEDES,DATEACQ,COMMENT)')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDL1 #JSMXCMD)
To_Get(#JSMSTS #JSMMSG #SKILLTBL)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDL1)
Else
Message Msgtxt('Employee skills not found')
Endif
Execute Subroutine(footer)
#rep1page += 1
Endselect
```

```
If (#outsts = OK)
Execute Subroutine(CLOSE)
Else
Message Msgtxt('No output produced')
* Close the JSM
Use Builtin(JSMX_CLOSE) With_Args(#jsmxhdl1) To_Get(#JSMSTS
#JSMMSG)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDL1)
Endif
```

* Produce PDF document for Employees by Department

```
When (= D)
```

```
#rep1page := 1
```

```
#outsts := *blanks
```

```
Select Fields(*all) From_File(pslmst1) With_Key(#deptment)
```

```
Nbr_Keys(*compute)
```

```
* Add Logo
```

```
Change Field(#JSMXCMD) To('ADD CONTENT(LOGO)')
```

```
Use Builtin(JSMx_COMMAND) With_Args(#JSMXHDL1 #JSMXCMD)
```

```
To_Get(#JSMSTS #JSMMSG)
```

```
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDL1)
```

* Clear employee header list

Clr_List Named(#header)

#fullname := #surname + ', ' + #givenname

Fetch Fields(#deptdesc) From_File(deptab) With_Key(#deptment)

Fetch Fields(#secdesc) From_File(sectab) With_Key(#deptment #section)

Add_Entry To_List(#header)

If_Status Is(*OKAY)

Change Field(#JSMXCMD) To('ADD CONTENT(HEADER)')

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD
#HEADER) To_Get(#JSMSTS #JSMMSG)

Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)

#outsts := OK

Else

Message Msgtxt('Employees not found for Dept ' + #deptment)

Leave

Endif

* Add Employee Skills List

* Get employee skills

* =====

Clr_List Named(#skilltbl)

Select Fields(#skills) From_File(pslskl) With_Key(#empno)

Nbr_Keys(*compute) Val_Error(*next)

Fetch Fields(#skills) From_File(skltab) With_Key(#skilcode)

Val_Error(*next)

#std_text := #dateacq.asdate(DDMMYY).asdisplayString(DDsMMsCCYY)

Execute Subroutine(grade)

Add_Entry To_List(#skilltbl)

Endselect

If_Status Is_Not(*error)

Change Field(#JSMXCMD) To('ADD CONTENT(SKILLTBL)

SERVICE_LIST(SKILDESC,GRADEDES,DATEACQ,COMMENT)')

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD)

To_Get(#JSMSTS #JSMMSG #SKILLTBL)

Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)

Else

Message Msgtxt('Employee skills not found')

Endif

```
Execute Subroutine(footer)
#rep1page += 1
Endselect
If (#outsts = OK)
Execute Subroutine(CLOSE)
Else
* Close the JSM
Use Builtin(JSMX_CLOSE) With_Args(#jsmxhdle1) To_Get(#JSMSTS
#JSMMSG)
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
Message Msgtxt('No output produced')
Endif
```

Personnel System

A business has a very simple Personnel System. The Personnel System allows the company to identify the employees in the company based on the part of the company where the employee works. The Personnel System lists details about the employees and details about their specific skills.

The company has a simple organizational structure. It is divided into departments such as Administration, Audit, Information Services, Legal, Travel, etc. Each of these departments may have one or more sections such as Accounting, Purchasing, Sales, etc. The Department table (DEPTAB) stores the list of departments. The Section table (SECTAB) is used to store the sections within each department.

The Personnel Master file (PSLMST) stores details about each employee. For example, the employee's name, address, and telephone number are stored in this master file. As each employee works in a section of a department, this information is also stored in the Personnel Master file.

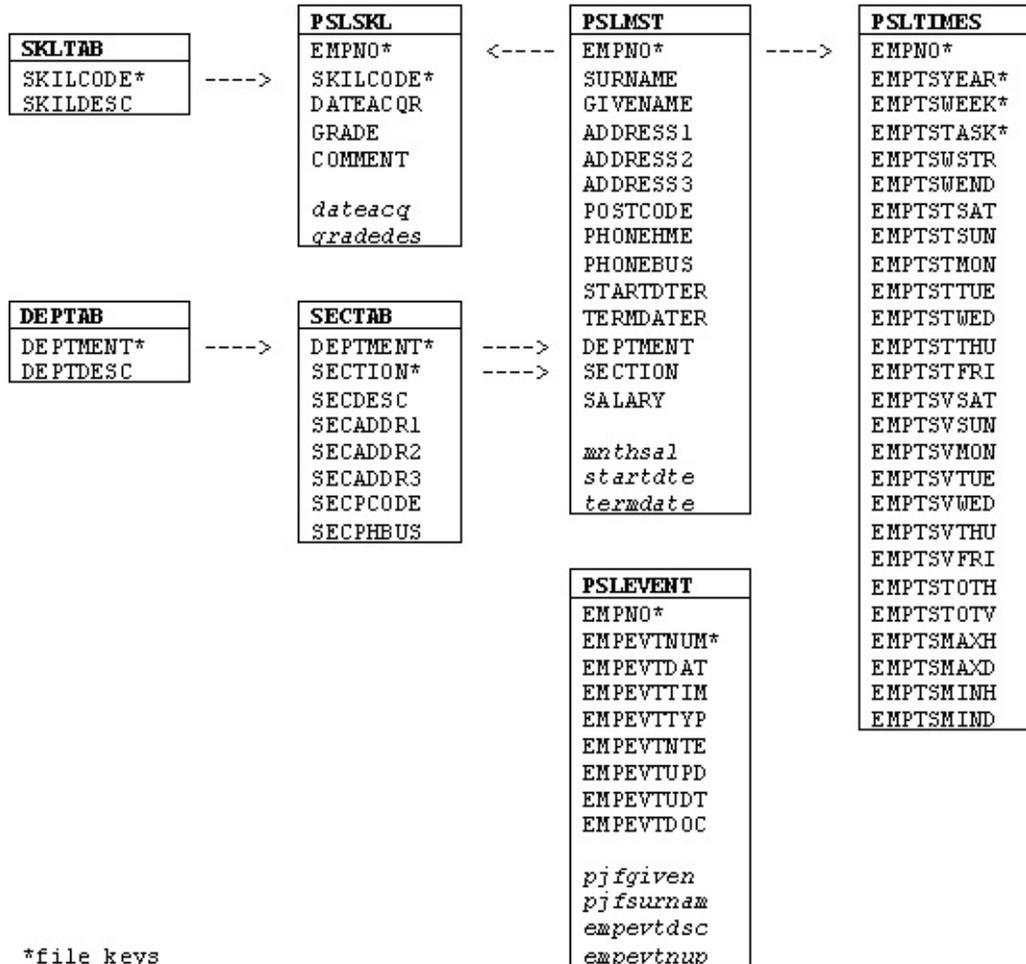
Each employee also has a list of skills. For example, an employee might have Cobol, C and C++ programming skills or management and administration skills. A Skills table (SKLTAB) is used to store the skill codes. A Personnel Skills file (PSLSKL) stores the specific skills of each employee.

The Personnel System is a very simple system. It has 5 files as described above. The physical database layout is described in Physical Database Map of Personnel System.

Historical Note: This system was created in 1987 as one of the very first LANSA demonstration and training systems. The LANSA repository and RDMLX functions created for this original system have been used on a System 38, AS/400, Windows, AIX, HPUX and other platforms. This original system has been left virtually unchanged to show how LANSA has been able to protect your investment in your application systems.

Physical Database Map of Personnel System

(Including Virtual and Predetermined Join Fields)



Sample Data in the Personnel Files

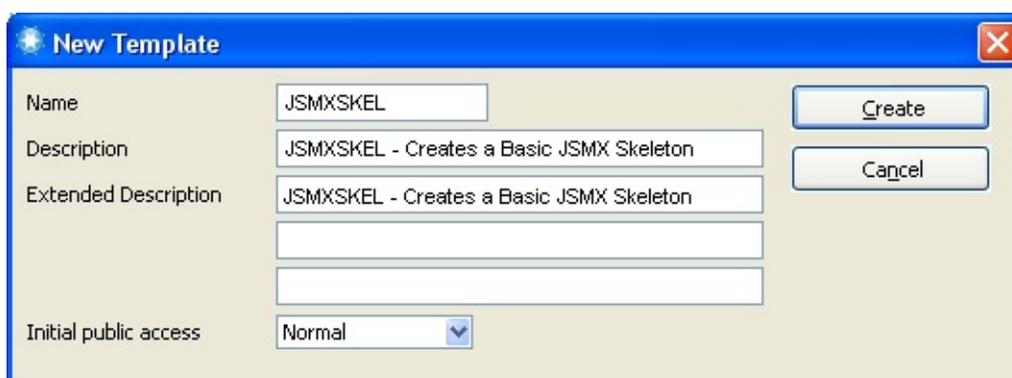
Following is a list of some of the sample data in the Personnel File which may be contained in the files. As developers edit these files, the data may have been altered:

DEPTAB:	SECTAB:	PSLMST:
DEPARTMENT	DEPARTMENT/SECTION	EMPNO
ADM	ADM 01	17 employees
	ADM 02	A1002
		A1005
		A1014
		A8888
	ADM 03	
	ADM 04	
	ADM 05	
AUD	AUD 01	
	AUD 02	
	AUD 03	
FLT	FLT 01	
	FLT 02	
	FLT 03	
INF	INF 01	
	INF 02	
	INF 03	

Create RDMLX Templates

The JSMXSKELE template is used in [Step 1. Create a Function using Template JSMXSKELE](#) of exercise [INT002 - Getting Started with Basic JSM Operations](#). Before you can do this exercise, you will need to create the JSMXSKELE template using the following steps.

1. Start the Visual LANSAs Development Environment, logon and select the partition that you will be using for these tutorials. Remember that the partition needs to be RDMLX enabled.
2. From the File menu, select *New* and then select *Template*.
3. Enter the details as shown in the following New Template dialog:



The screenshot shows a 'New Template' dialog box with the following fields and values:

Field	Value
Name	JSMXSKELE
Description	JSMXSKELE - Creates a Basic JSMX Skeleton
Extended Description	JSMXSKELE - Creates a Basic JSMX Skeleton
Initial public access	Normal

4. Press *Create* when done and the *Source* tab will be opened for the new template.
5. Copy the code supplied in [JSMXSKELE Source Code](#) and paste it into the open *Source* tab.
6. *Save* it.
7. Create a second new template in the same way as for JSMXSKELE.
8. Enter the template's details in the *New Template* dialog as shown in the following diagram:

New Template

Name: BBJSMXCMD

Description: BBJSMXCMD - Building Block - Build a JSM

Extended Description: BBJSMXCMD - Building Block - Build a JSM

Initial public access: Normal

Buttons: Create, Cancel

9. Press *Create* when done and the *Source* tab will be opened for the new template.
 10. Copy the code supplied in [BBJSMXCMD Source Code](#) and paste it into the open *Source* tab.
 11. *Save* it.
You will now see your two new templates in the template list for this partition.
- Return to the exercises.

JSMXSKEL Source Code

```
@@COMMENT '=====
@@COMMENT ' Process .....: @@PROCESS '
@@COMMENT ' Function .....: @@FUNCTION '
@@COMMENT ' Created on .....: @@DATE at @@TIME '
@@COMMENT ' Description ....: @@FUNCDES '
@@COMMENT ' Template.....: JSMXSKEL '
@@COMMENT '=====
FUNCTION OPTIONS(*DIRECT)
@@COMMENT '
@@COMMENT ' OPEN JSM AND VERIFY STATUS '
USE BUILTIN(JSMX_OPEN) TO_GET(#JSMSTS #JSMMSG #jsmxhdl
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#jsmxhdl1)
@@COMMENT '
@@QUESTION PROMPT('Do you wish to load a JSM Service?') ANSWER(
@@IF COND((*IF @@CANS001 *EQ ' ')) GOTO(L01)
@@COMMENT 'BUILD THE SERVICE LOAD COMMAND '
#JSMXCMD := 'SERVICE_LOAD'
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(#JSMXCMD 'SERV
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSM
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXHDLE1)
L01: @@LABEL
@@COMMENT '
@@COMMENT ' YOUR OWN LOGIC HERE '
@@COMMENT '
@@IF COND((*IF @@CANS001 *EQ ' ')) GOTO(L02)
@@COMMENT 'UNLOAD SERVICE
#JSMXCMD := 'SERVICE_UNLOAD'
USE BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSM
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXHDLE1)
L02: @@LABEL
@@COMMENT 'CLOSE JSM AND VERIFY STATUS '
USE BUILTIN(JSMX_CLOSE) WITH_ARGS(#JSMXHDLE1) TO_GET
EXECUTE SUBROUTINE(CHECK_STS) WITH_PARMS(#JSMXHDLE1)
@@COMMENT '
RETURN
@@COMMENT '

```

```

@@COMMENT 'Subroutine to build JSM commands. existing JSM comman
@@COMMENT '
SUBROUTINE NAME(KEYWRD) PARMS((#W_CMDX *BOTH) (#W_KEY
DEFINE FIELD(#W_CMDX) REFFLD(#JSMXCMD)
DEFINE FIELD(#W_KEYWRD) REFFLD(#STD_TEXT)
DEFINE FIELD(#W_KEYVAL) REFFLD(#STD_TEXTL)
#W_CMDX += ' ' + #W_KEYWRD + '(' + #W_KEYVAL + ')'
ENDROUTINE
@@COMMENT '
@@COMMENT ' Check the status of the JSM command issued'
@@COMMENT '
SUBROUTINE NAME(CHECK_STS) PARMS(#W_HDLE)
@@COMMENT '
DEFINE FIELD(#MSGDTA) TYPE(*CHAR) LENGTH(132)
DEFINE FIELD(#W_HDLE) TYPE(*CHAR) LENGTH(4)
@@COMMENT '
IF COND('#JSMSTS *NE OK')
@@COMMENT '
#MSGDTA := 'Error Status Code: ' + #JSMSTS
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
#MSGDTA := 'Error Message: ' + #JSMMSG
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#MSGDTA)
ENDIF
@@COMMENT '
ENDROUTINE

```

BBJSMXCMD Source Code

```
@@QUESTION PROMPT('Use this template to build a command in a functio
@@COMMENT 'BUILD THE JSM COMMAND '
L01: @@LABEL
@@QUESTION PROMPT('Type in the JSM command you want to build') AN
#JSMXCMD := @@CANS001
@@QUESTION PROMPT('How many keywords do you want to specify?') A
values. ' 'Otherwise, type in a number of up to 20 keyword-
value to be prompted') LOWER(*NO) RANGE((0 20))
@@IF COND((*IF @@NANS001 *EQ 0)) GOTO(L05)
@@SET_IDX IDX_NAME(II) TO(1)
LOP: @@LABEL
@@QUESTION PROMPT('Type in the keyword name.') ANSWER(@@CAN
@@QUESTION PROMPT('Type in the value for the @@CANS002 keyword'
@@IF COND((*IF @@CANS003 *EQ ' ')) GOTO(L02)
@@IF COND((*IF @@CANS002 *EQ 'URI')) GOTO(L07)
@@GOTO LABEL(L08)
L07: @@LABEL
IF COND('#CPUTYPE = AS400')
#STD_TEXTL := '/CGI-BIN/JSMDIRECT?' + @@CANS003
ELSE
#STD_TEXTL := '/CGI-BIN/JSMDIRECT.EXE?' + @@CANS003
ENDIF
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD @@C
@@GOTO L03
L08: @@LABEL
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD @@C
@@GOTO L03
L02: @@LABEL
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(#JSMXCMD @@C
L03: @@LABEL
@@INC_IDX IDX_NAME(II)
@@CMP_IDX IDX_NAME(II) IDX_VALUE(@@NANS001) IF_GT(L04)
@@GOTO LABEL(LOP)
L04: @@LABEL
@@QUESTION PROMPT('Type in the name of a working list if this comman
@@IF COND((*IF @@CANS004 *EQ ' ')) GOTO(L05)
```

```
USE      BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSM
EXECUTE  SUBROUTINE(CHECK_STS) WITH_PARM(#JSMXCMD)
@@COMMENT ' '
@@GOTO   LABEL(L06)
L05: @@LABEL
USE      BUILTIN(JSMX_COMMAND) WITH_ARGS(#JSMXHDLE1 #JSM
EXECUTE  SUBROUTINE(CHECK_STS) WITH_PARM(#JSMXCMD)
@@COMMENT ' '
L06: @@LABEL
@@QUESTION PROMPT('Do you wish to enter another JSM command?') A
@@IF     COND((*IF @@CANS005 *EQ Y)) GOTO(L01)
END: @@LABEL
/* ***** */
@@COMMENT ' '

```

Appendix A. Performance and Tuning

[Java Service Manager Thread Safety](#)

[IBM i - Java Heap Size](#)

[IBM i - Pool Size](#)

[IBM i – Work with JVM Jobs](#)

[IBM i - Tuning GC for Java](#)

[IBM i - DUMP JVM](#)

[IBM i - ANALYZE JVM](#)

[IBM i - Network Performance](#)

[IBM i - Java License Program](#)

[Oracle – Java Performance Tuning](#)

[JSM Console](#)

Java Service Manager Thread Safety

Java Service Manager is thread-safe in the context that each JSM client program has its own thread and each object instance is only visible to this thread.

The few internal resources that have shared access have been enclosed within synchronized blocks.

The trace sequence number is a JVM global shared incrementing integer and is accessed once or twice by a thread during its lifetime. Once a thread has a unique number it does not need to access this shared data again. If tracing is off, then a thread never needs to access this synchronized resource.

All objects passed to the JSMService class are new and visible only to the current thread.

- JSMClient
- JSMTrace
- JSMContainer
- JSMStorage
- JSMResource
- JSMCommand
- JSMField
- JSMFieldList
- JSMList
- JSMResponse
- JSMException

The JSMManager class has two static methods: createTemporaryFile and clearTemporaryDirectory which create and delete files from the IFS.

The IFS is a shared resource to all threads running within the same JVM job and all other jobs running on the machine. It adds no value to synchronize these methods, because of the bottleneck it would create for threads trying to create and clear temporary files. Each thread has a unique IFS subdirectory to use for temporary storage and no resource contention will occur.

Being only visible to the current thread means that no other thread can read and write to these objects, unless the service programmer creates a new thread and passes an object reference to this new thread.

Most of the JSM classes have instant variables that are private and final, so their

values cannot change after construction. All JSM classes internally use objects that have synchronized methods.

Normally only one thread is ever accessing the state of a JSM service object and these synchronized methods only add a lock and unlock overhead and do not effect the concurrency of other threads.

If another thread has access to an object reference, then these synchronized methods would then protect the shared data.

JSMClient instance variables are private and final and cannot be changed. This object is thread-safe because it is read-only and the internal state will not change.

JSMContainer instance variables are private JSMTrace, JSMStorage and JSMResource objects.

The JSMStorage object reference never changes.

The JSMTrace and JSMResource object references change when a SERVICE_LOAD command is received.

JSMTrace instance variables are private and cannot be changed and its print methods are synchronized. When the SERVICE_LOAD command is received a new JSMTrace object is created. The current JSMTrace object is closed and is no longer usable.

JSMStorage instance variable is a private and final Hashtable object. The object reference cannot be changed but the Hashtable contents can be modified by the synchronized get and put methods.

JSMResource instance variable is a private and final Hashtable object. The object reference cannot be changed but the Hashtable contents can be modified by the synchronized get and put methods.

JSMField instance variables are private and final String and DataType objects and cannot be changed.

DataTypePacked and DataTypeZoned instance variables are private and final integers and cannot be changed.

DataTypeText instance variables are private and final String, integer and byte array objects and cannot be changed.

JSMCommand instance variables are private and final and cannot be changed. This means that the internal object references will not change but you need to check the documentation on the internal behaviour of these objects.

JSMCommand is just a holder object for a collection of other objects.

The JSMFieldList and JSMList objects are **not thread-safe**. There are no

synchronized methods to protect the internal state of any instance variables. If more than one thread needs to change the contents of these objects then they will need to be externally synchronized by the programmer.

JSMResponse is **not thread-safe**. There are no synchronized methods to protect the internal state of an instance variables and therefore cannot be made visible to other threads of work.

JSMException instance variables are private and set at construction time and cannot be changed.

IBM i - Java Heap Size

JAVA command keywords:

- GCHINL- Initial garbage collection heap size, in kilobytes
- GCHMAX- Maximum heap size, in kilobytes

The GCHINL value determines how the garbage collector runs. Each time memory usage equal to the GCHINL value has been consumed, the garbage collector will be invoked. A large value makes the GC run less frequently, but for a long time.

The Java heap size settings determine the amount of memory used by the Java Virtual Machine. Note that these settings behave differently on IBM i than on other platforms.

On IBM i, the initial Java heap size specifies a threshold for the amount of memory in the JVM; once that amount of memory has been allocated, the garbage collector will begin to run.

Unlike most other JVMs, the IBM i JVM allows other threads to continue running while the garbage collector is reclaiming unused memory.

The initial heap size should be tuned so that the garbage collector runs at a suitable interval -- infrequently enough that it does not cause unnecessary overhead, yet often enough that the heap does not grow too large.

The GCHMAX value determines the maximum heap size of the JVM, the default value is *DFT.

The GCHMAX *DFT value for the IBM Classic JVM is *NOMAX.

The GCHMAX *DFT value for the IBM Technology for Java JSM is 2G.

This default value of 2G is too small for a 64-bit Java environment and needs to be increased.

Do not use the maximum value of 240000000 as the IBM Technology for Java command treats this as a *NOMAX value and defaults back to the 2G value.

From experimentation, these are the maximum GCHMAX values for IBM Java Technology for Java JDK's.

32bit JVM's are limited to 3407872 kilobytes

```
STRJSM VERSION(*JVM5032) GCHMAX(3407872)
```

64bit JVM's are limited to 239861760 kilobytes

```
STRJSM VERSION(*JVM5064) GCHMAX(239861760)
```

If you are using the shared classes option (-Xshareclasses), then the maximum GCHMAX value will be less.

Refer to the following IBM Redbook:

<http://www.redbooks.ibm.com/abstracts/sg247353.html>

IBM i - Pool Size

Java is an object oriented language, so it creates objects in storage, Generally a lot of objects.

What is more, Java provides automatic garbage collection (GC), which means that some time after an object is no longer able to be accessed by any running Java code, the Garbage Collector will make sure it is deleted.

The IBM i server allows the amount of real (as opposed to virtual) memory to be specified by a Memory Pool.

Page faults occur whenever the actual size of your GC heap, and any other programs using memory from that pool if the pool is shared, exceed the amount of real memory specified in the pool.

Excessive page faults may occur if the memory pool for your JVM is too small. These faults will be reported as non-database page faults on the WRKSYSSTS command display.

Typically, the storage pool for your JVM is *BASE.

Fault rates between 20 and 30 per second are generally acceptable, but higher rates should be reduced by increasing memory pool size.

In some cases, reducing this value below 20 or 30 per second may improve performance as well.

Lowering the GCHINL parameter might also reduce paging rates by reducing the i5/OS JVM heap size, but may also cause some performance problems due to more frequent GC cycles.

A memory pool also has an activity level associated with it which specifies the number of threads that can actively use processor(s) at the same time from that memory pool.

When more threads are started than are allowed to concurrently execute due to the activity level control, the excess threads will be forced to wait for an available activity level slot before they can run.

The number of threads running (active threads) refers to the number of threads that are eligible to compete for a processor and that count against the activity level for a memory pool.

Active threads do not include threads that are waiting for input, for a message, for a device to be allocated, or for a file to be opened.

Active threads do not include threads that are ineligible (threads that are ready

to run but the memory pool activity level is at its maximum).

Once the maximum activity level for a memory pool has been reached, additional threads needing the memory pool are placed in the ineligible state.

The threads wait in ineligible state for the number of active threads in the memory pool to fall below the maximum activity level, or for a thread to reach the end of its time slice.

As soon as a thread gives up its use of the memory pool, the other threads that are not active become eligible, and will be dispatched based on their priority.

Having Java threads in the ineligible state can cause severe performance degradation as well as excessive JVM GC heap growth.

When the Java threads are in the ineligible state they are unable to communicate with the Garbage Collector, which will prevent a Java GC cycle from completing.

This will cause the GC heap to grow rapidly, which will tend to drive the pool into page thrashing, and further degrade performance.

In order to avoid this condition, set the activity level of the memory pool to at least as large as the maximum number of threads you expect to be concurrently active at any time for the JVM running in the memory pool.

IBM i – Work with JVM Jobs

You can use the Work with JVM Jobs (WRKJVMJOB) CL command to collect performance data.

You can access the information available from the WRKJVMJOB command from both the Work with Job (WRKJOB) screen as well as by issuing the WRKJVMJOB command.

The following information or functionality is available when using WRKJVMJOB:

- The arguments and options with which the JVM was started.
- Environment variables for both ILE and PASE for i.
- Java lock requests outstanding for the JVM job.
- Garbage collection information.
- Java system properties.
- The list of threads associated with the JVM.
- The partially completed job log for the JVM job.
- The ability to work with spooled input and output files for the JVM job.
- The ability to generate JVM (system, heap, Java) dumps from a panel option. These capabilities are also available from the Generate JVM Dump (GENJVMDMP) command.
- The ability to enable and disable verbose garbage collection from a panel option.

IBM i - Tuning GC for Java

At times, it may be helpful to collect additional data about the garbage collector as it's running.

This can be accomplished with verbose GC, which dumps information each time the collector runs.

This includes the current heap size, as well as the number and size of objects collected, number of objects in the heap, amount of time the collector ran and other information.

Sample Verbose GC Output:

```
GC 4: starting collection, threshold allocation reached.  
GC 4: live objects 2562187; collected objects 4936351; collected (KB) 541840  
GC 4: queued for finalization 0; total soft references 92; cleared soft references  
GC 4: current heap (KB) 1171424; current threshold (KB) 524288.  
GC 4: collect (milliseconds) 4138.  
GC 4: current cycle allocation (KB) 236160; previous cycle allocation (KB) 52  
GC 4: total weak references 684; cleared weak references 0.  
GC 4: total final references 11797; cleared final references 63.  
GC 4: total phantom references 0; cleared phantom references 0.  
GC 4: total old soft references 0; cleared old soft references 0.
```

The most important of these fields for tuning GC are:

- Memory allocated since the last cycle began Previous cycle allocation (KB).
- Memory allocated since the current cycle began Current cycle allocation (KB).
- Elapsed time for this cycle Collect (milliseconds).
- The threshold value Current threshold (KB).
- The current heap size Current heap (KB).
- Total size of the objects collected during this cycle Collected (KB).
- Number of objects collected during this cycle Collected objects.
- Number of objects currently active in the JVM Live objects.
- This is the fourth GC cycle since the JVM started GC 4.

The current threshold is the value set for the initial heap size (512 MB in the example output).

The previous cycle allocation is normally close to this value, because the GC cycle is triggered when the amount of memory allocated since the last cycle began reaches the threshold value.

The example output shows that the GC cycle took more than four seconds to complete. During that time, the current cycle allocation reached more than 200 MB. This is about 40 percent of the threshold value, which suggests that the total time between the beginning of this cycle and the next cycle is around 10 seconds. This cycle collected nearly 5 million objects (collected objects), leaving only 2.5 million objects in the heap at the end of the cycle (live objects).

In general, it's best to have a low cycle time. One to two seconds is ideal, but times of five to 10 seconds are common.

It's also best to have some time between collection cycles (i.e., current cycle allocation should be less than the current threshold).

These two goals work against each other -- increasing the threshold value allows the heap to grow, resulting in more time between cycles, but this lengthens each cycle. Decreasing the threshold shortens each cycle, but also shortens the time between cycles.

The key to tuning GC is to find a balance between these two goals. This is why examining the CPU consumed by GC is generally better than looking at values like the current heap size.

If you don't have the Performance Tools licensed program product (PT1 LPP) installed on your system, you may want to experiment with the threshold value, using your application's throughput or response time as well as verbose GC output to determine how to further tune the threshold.

Even if you do have PT1, it may be useful to examine verbose GC output to understand how your application uses the heap and to watch for changes in the GC behavior as application and system loads change.

IBM i - DUMP JVM

Another tool that can be used to learn about garbage collector performance is the Dump JVM (DMPJVM) command, which provides a spool file with information about your JVM, including some of the key GC data -- initial heap size, maximum heap size, current heap size and number of collections since the JVM was started.

It also includes a dump of the objects currently in the heap, which can be helpful for analyzing object leak problems. (Object leaks occur when your application creates new objects and keeps a reference to the objects even when they're no longer needed, preventing the collector from collecting them.)

Sample of the GC section of DMPJVM output:

```
Garbage collector parameters
Initial size: 262144 K
Max size: 240000000 K
Current values
Heap size: 449952 K
Garbage collections: 278
Additional values
JIT heap size: 85728 K
JVM heap size: 186588 K
Last GC cycle time: 1302 ms
```

While the DMPJVM data is only a snapshot and doesn't provide the details available with verbose GC, it can be run without restarting your JVM, so it's useful for getting some information about the JVM after a problem has occurred.

Details on DMPJVM can be found in the IBM i Information Center at: <http://publib.boulder.ibm.com/iseres/v5r2/ic2924/info/cl/dmpjvm.htm>).

Use WRKACTJOB command to locate the JSM JVM job.

```
JSM
JSMJOB      QOTHPRDOWN  BCH  .0  PGM-RUNJSM  TIMW
QJVACMDSRV QOTHPRDOWN  BCI  .0                TIMW
```

Job: QJVACMDSRV User: QOTHPRDOWN Number: 422841

Use DMPJVM command to create a spool file with the JVM dump information.

DMPJVM JOB (422841/QOTHPRDOWN/QJVACMDSRV)

Sample DMPJVM on the JSM JVM:

Java Virtual Machine Information 422841/QOTHPRDOWN/QJVACMDSRV

.....

Classpath

.....

java.version=1.2

/QIBM/ProdData/Java400/jdk12/lib/jdkptf12.zip:/QIBM/ProdData/Java400/jdl

.....

Garbage Collection

.....

Garbage collector parameters

Initial size: 2048 K

Max size: *NOMAX

Current values

Heap size: 44032 K

Garbage collections: 51

.....

Thread information

.....

Information for 4 thread(s) of 4 thread(s) processed

Thread: 00000001 Thread-0

TDE: B000100007CCA000

Thread priority: 5

Thread status: Running

Thread group: main

Runnable: java/lang/Thread

Stack:

java/net/PlainSocketImpl.accept(Ljava/net/SocketImpl;)V+1 (PlainSocketImpl

java/net/ServerSocket.implAccept(Ljava/net/Socket;)V+36 (ServerSocket.java

java/net/ServerSocket.accept()Ljava/net/Socket;+8 (ServerSocket.java:224)

com/lansa/jsm/d.if()Lcom/lansa/jsm/a;+0 (:0)

com/lansa/jsm/JSMMManager.do(Ljava/lang/String;)V+0 (:275)

```
com/lansa/jsm/JSManager.main([Ljava/lang/String;)V+0 (:311)
Locks:
None
Thread: 00000002 Reference Handler
TDE: B0001000070D4000
Thread priority: 10
Thread status: Waiting
Wait object: java/lang/ref/Reference$Lock
Thread group: system
Runnable: java/lang/ref/Reference$ReferenceHandler
Stack:
java/lang/ref/Reference$ReferenceHandler.run()V+48 (Reference.java:129)
Locks:
None
Thread: 00000003 Finalizer
TDE: B000100007CCE000
Thread priority: 8
Thread status: Waiting
Wait object: java/lang/ref/ReferenceQueue$Lock
Thread group: system
Runnable: java/lang/ref/Finalizer$FinalizerThread
Stack:
java/lang/ref/ReferenceQueue.remove(J)Ljava/lang/ref/Reference;+48 (Refere
java/lang/ref/Finalizer$FinalizerThread.run()V+3 (Finalizer.java:190)
Locks:
None
```

Once you've picked a reasonable starting point, start your application and let it run for a while under the maximum load that you intend to handle, giving it time to reach a steady state (a few minutes is usually sufficient). It's best to use a load-generation tool to put a constant load on your system. This allows you to tune your application in a development environment rather than a production environment.

In addition to providing a constant load (allowing you to see the effects of changes more accurately), this allows you to make changes as necessary without affecting users.

While your application is running, use the aforementioned tools to measure the impact of your changes. Your load-generation tool should also detail the

throughput and response time so you can see the impact on your application's performance.

Whenever you change performance parameters such as the GC threshold, you should measure the effects on throughput and/or response time to ensure that the changes actually help. If the tools indicate that there's still room for improvement, change the threshold and try a new run. Once you've increased it too far, the throughput begins to degrade again, indicating that you must reduce the threshold.

These recommendations are based on the assumption that your system has enough main storage to handle larger heap sizes. In reality, this may not always be the case.

In systems with limited memory, it may be necessary to set the GC threshold to a lower value, which increases collection frequency and decreases heap size. This allows the entire heap to be kept in memory.

Use Work with System Status (WRKSYSSTS) command to monitor the non-database paging/faulting rates.

If these rates get too high, the heap may be too large. The definition of "too large" depends on a variety of factors--system size, number of disks and system workload--but sustained non-database paging rates greater than 10 faults per second by Java programs is generally cause for concern.

Higher paging rates are acceptable during "warm-up" periods.

High paging rates may result from having the GC threshold set too high or may be a symptom of a larger problem.

In this case, the first step should be to isolate the JVM in its own memory pool.

This reduces the effects that other applications may have on the JVM and makes it easier to identify whether the problem is with the GC settings, system configuration or simply not enough hardware to handle the workload.

In cases where memory is especially limited, it may be useful to set the maximum heap size.

Normally, this should be left at the default value *NOMAX, which means that GC runs only when the GC threshold has been reached.

If a maximum heap size is set, the collector runs whenever the heap reaches that maximum size.

However, unlike a normal GC, if the maximum size is reached, all application threads must wait until the collector has finished before they can continue running. This results in undesirable pause times.

Therefore, it's preferred to use the maximum heap size as a safety net to handle times of unexpected heap growth and ensure that the heap doesn't grow larger than the available memory.

The GC threshold should be set so that this maximum size is never actually reached under normal circumstances.

Tuning the collector is one way to reduce the amount of time spent in GC. Another way is to reduce the number of objects being created. (Note: Tips for reducing object creation are common to all platforms and documented in several places.

IBM i - ANALYZE JVM

The Analyze Java Virtual Machine (ANZJVM) command collects information about the Java Virtual Machine (JVM) for a specified job.

A set of JVM information is collected immediately when the command is run. This collected JVM data is called a snapshot.

A second snapshot is done a specified amount of time later.

By taking a snapshot of the JVM and comparing the data with a snapshot taken at a later time the data can be analyzed to help find object leaks.

To find object leaks, you would look at the number of instances of each class in the heap. Classes that have an unusually high number of instances should be noted as possibly leaking.

You should also note the change in number of instances of each class between the two copies of the garbage collection heap. If the number of instances of a class continually increases, that class should be noted as possibly leaking.

The longer the time interval between the two copies, the more certainty you have that objects are actually leaking.

By running ANZJVM a series of times with a larger time interval, you should be able to diagnose with a high degree of certainty what is leaking.

The information is dumped using printer device file QSYSPRT.

The user data for the QSYSPRT file is 'ANZJVM'.

The dump includes formatted information about the JVM heap.

Details include names of classes, number of active objects per class, and the class loader used to load each class.

When you run the ANZJVM command, there is a parameter to specify if a garbage collection cycle should be forced, and if so, an attempt is made to force a garbage collection cycle before each pass.

There are also parameters on how the information should be stored and how long the interval between passes should be.

The following ANZJVM command example collects two copies of the JVM 60 seconds apart for a job named QJVACMDSRC for user name QOTHPRDOWN and the job number 099112.

```
ANZJVM JOB (099112/ QOTHPRDOWN /QJVACMDSRV)
```

IBM i - Network Performance

Network Tuning

Network performance is influenced by the Maximum Transmission Unit size, send and receive buffer size, port filtering, duplex settings and Domain Name lookup.

The Maximum Transmission Unit Size (MTU) parameter affects the actual size of the line flows.

By increasing the value of this parameter you can reduce the overall number of transmissions, and therefore, increase the potential capacity of the CPU and the IOP (input/output processor).

Similar parameters also exist on the client.

The negotiated value will be the minimum of the server and client (and perhaps any bridges/routers), so increase them all.

Setting the Maximum Transmission Unit

The maximum transmission unit (MTU) parameter is available on the following commands.

Add TCP/IP Interface (ADDTCPIFC)

Add TCP/IP Route (ADDTCPRTE)

Change TCP/IP Interface (CHGTCPIFC)

Change TCP/IP Route (CHGTCPRTE)

Following is a list of the maximum MTU values that you can specify, based on the line type:

Asynchronous (SLIP)	1006
DDI	4352
Ethernet 802.3	1492
Ethernet Version 2	1500
Frame relay	8177
Point-to-Point (PPP)	4096
Token ring (4 meg)	4060

Token ring (16 meg)	16388
Wireless 802.3	1492
Wireless Version 2	1500
X.25	4096

TCP/IP processing uses a small part of each datagram. Therefore, the whole datagram size is unavailable for user data.

The value of the maximum transmission unit used by TCP/IP processing depends on the value that you specify for the route on the MTU parameter of the route or interface commands mentioned previously. It also depends on the type of physical line that you use, the maximum frame size of the network line, and the SSAP maximum frame size.

Determining the Maximum Size of Datagrams

For a communications line, specify the maximum frame size on the appropriate Create Line Description command.

The maximum frame size is compared to the MTU value of the route or interface.

TCP/IP uses the lesser of these two values to determine the maximum size of datagrams that it sends by over this line.

For example, if you specify 1024 for the MTU parameter for a route attached to a communications line and the line description contained a value of 512 for a maximum frame size, the maximum datagram size value for the route that TCP/IP uses is 512.

If the line is varied off and you change the maximum frame size on the Token-ring line description to 1994, and then the line is varied on, the maximum transmission unit used for the route is reset to 1024 when the next TCP/IP operation occurs that causes a datagram to be sent.

Display TCP/IP Route

Route destination	*DFTRROUTE
Subnet mask	*NONE

Type of service	*NORMAL
Next hop	10.2.0.1
Preferred binding interface	*NONE
Maximum transmission unit	*IFC
Duplicate route priority	5
Route metric	1
Route redistribution	*NO

Display TCP/IP Interface

Internet address	10.2.0.169
Subnet mask	255.255.0.0
Line description	ETHLINE
Line type	*ELAN
Associated local interface	*NONE
Interface status	Active
Type of service	*NORMAL
Maximum transmission unit	1492
Automatic start	*YES

TCP/IP Buffer Size

Server performance can be improved by tuning the buffer size that is used by TCP/IP.

If your network is very reliable, try increasing the buffer size from the default value.

If your network experiences a significant amount of collisions or congestion, you may be able to improve performance by decreasing the TCP/IP send and receive buffers. This is because it will take less time to detect a bad packet, and less data will need to be re-transmitted.

Select a size in the range of 32K to 128K, and use the same size for both buffers.

Setting the buffer size to a large value, 8388608 for example, can cause storage allocation problems on your machine. This is because every TCP/IP connection allocates 8MB of storage for its receive buffer.

For best use of resources the buffer size should be a multiple of the amount of data a datagram can hold, this is called the maximum segment size (MSS).

To determine the MSS is problematic as the IP header and the TCP header may

vary in length. An optimistic position assumes that both the IP header and the TCP header are minimum size, that is, 20 octets each.

The value of the MSS is the MTU minus (20 + 20), which is $1492 - 40 = 1452$ bytes.

You need to make the buffers' size a multiple of the MSS.

For a buffer size of about 32K, you calculate it as follows: $32,000 / 1,452 = 22.04$.

Discard the decimal places in the result and multiply the whole number by the MSS.

$22 * 1,452 = 31,944$

That's the number to set your send and receive buffer sizes too.

Change TCP/IP Attributes

TCP keep alive	120
TCP urgent pointer	*BSD
TCP receive buffer size	31944
TCP send buffer size	31944
TCP R1 retransmission count	3
TCP R2 retransmission count	16
TCP closed timewait timeout	120
UDP checksum	*YES

Duplex

Ethernet supports both half and full duplex.

The best performance will be with full duplex.

However, the duplex setting on the line description must match the setting on the port on the switch if the line is hooked up to a switch.

Be especially careful if you set your ethernet switch or your line description to *AUTO. It has been found in many cases that performance is severely degraded because the duplex setting did not auto-negotiate correctly.

It is probably best to configure switches and line descriptions to either *FULL or *HALF explicitly.

If the line is connected to a 'stackable' hub, then the duplex setting on the line should be set to *HALF to match the hub setting.

Domain Name Server Lookup

TCP/IP connection performance can be improved if the registered DNS servers are available and performing well.

The JSM_OPEN Built-In Function performs a DNS lookup to resolve the host name address. Using a dotted decimal IP address does not get around this DNS lookup.

If the JSM_OPEN Built-In Function is slow to open a connection then you could have a DNS lookup issue. Add an entry to the local host table to improve performance.

```
Change TCP/IP Domain
Host name search priority      *LOCAL
Domain name server
Internet address              139.130.4.4
                              203.48.48.13
```

If the first Domain Name Server (DNS) in the list does not respond, the second DNS server in the list will be contacted. If the second DNS server does not respond, the third DNS server will be contacted.

IBM i - Java License Program

The IBM i platform supports multiple versions of the Java Development Kit (JDK).

If you are using IBM Technology for Java, you select which option to run by setting the JAVA_HOME environment variable.

This differs from the Classic JVM, which uses the java.version system property.

The following table lists the supported options:

Option	JAVA_HOME	Version
Option 6	/QIBM/ProdData/Java400/jdk14	Classic 1.4
Option 7	/QIBM/ProdData/Java400/jdk15	Classic 1.5 (5.0)
Option 8	/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit	IBM Technology for Java 5.0 32-bit
Option 9	/QOpenSys/QIBM/ProdData/JavaVM/jdk50/64bit	IBM Technology for Java 5.0 64-bit
Option 10	/QIBM/ProdData/Java400/jdk6	Classic 1.6 (6.0)
Option 11	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit	IBM Technology for Java 6 32-bit
Option 12	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit	IBM Technology for Java 6 64-bit
Option 13	/QOpenSys/QIBM/ProdData/JavaVM/jdk14/64bit	IBM Technology for Java 1.4 64-bit
Option 14	/QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit	IBM Technology for Java 7 32-bit
Option	/QOpenSys/QIBM/ProdData/JavaVM/jdk70/64bit	IBM Technology

15

for Java 7 64-bit

Option /QOpenSys/QIBM/ProdData/JavaVM/jdk80/32bit IBM Technology
16 for Java 8 32-bit

Option /QOpenSys/QIBM/ProdData/JavaVM/jdk80/64bit IBM Technology
17 for Java 8 64-bit

Note: If you install only one JDK, the default JDK is the one you installed.
If you install more than one JDK, consult IBM documentation on the order of
precedence.

Oracle – Java Performance Tuning

Oracle's JDK is used for the Windows and Red Hat Linux platform. For performance tuning refer to Oracle documentation.

[Java Tuning Java Tuning White Paper](#)

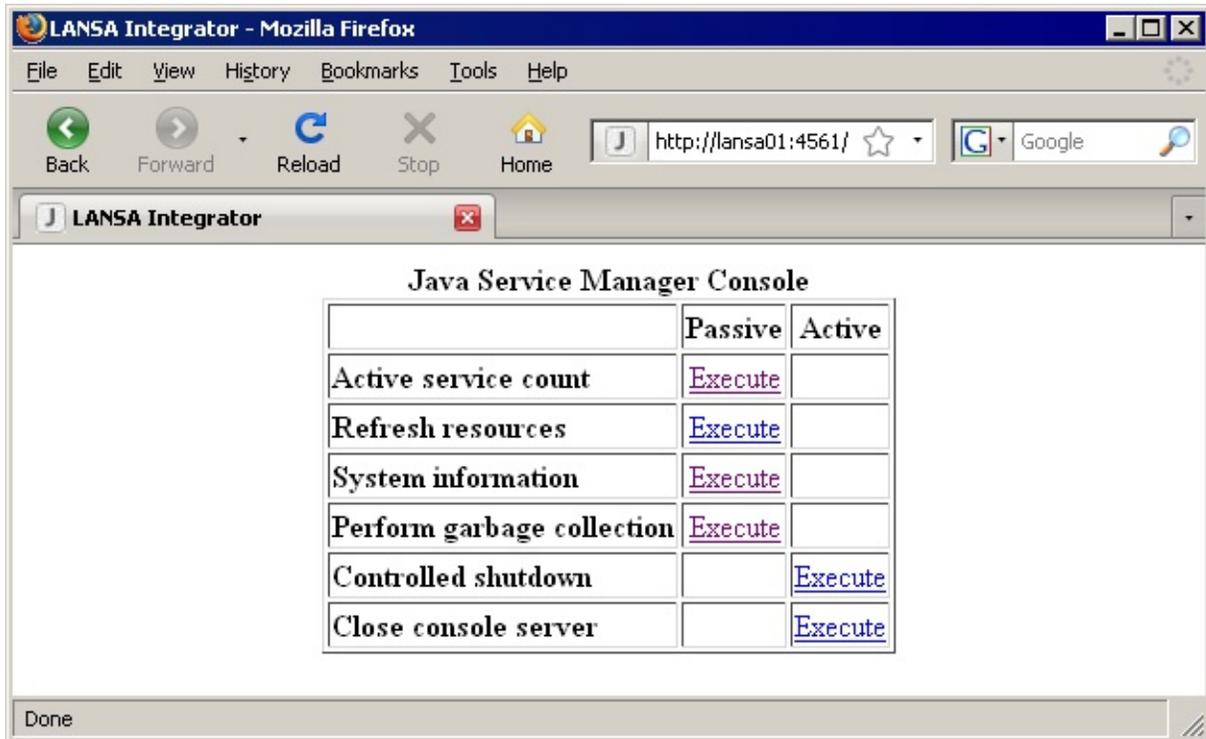
[Tuning Garbage Collection with the 5.0 Java™ Virtual Machine](#)

[Java SE 6 Performance White Paper](#)

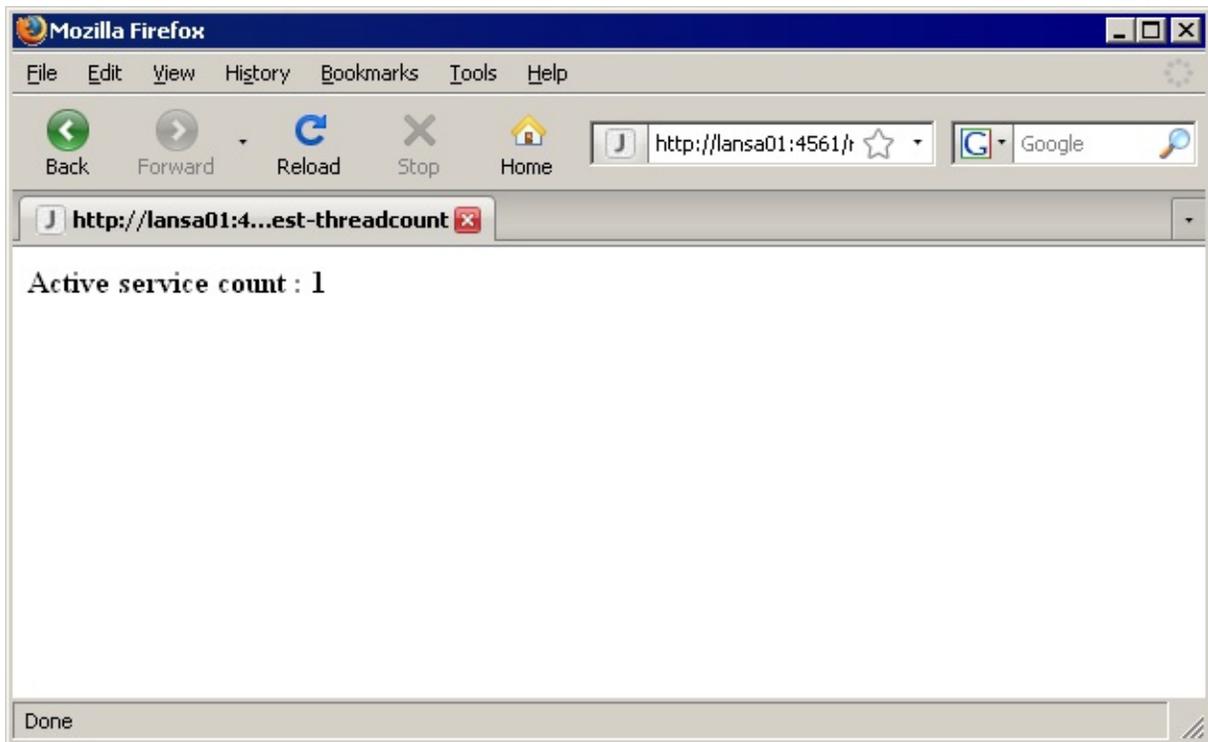
[Java SE 6 HotSpot™ Virtual Machine Garbage Collection Tuning](#)

JSM Console

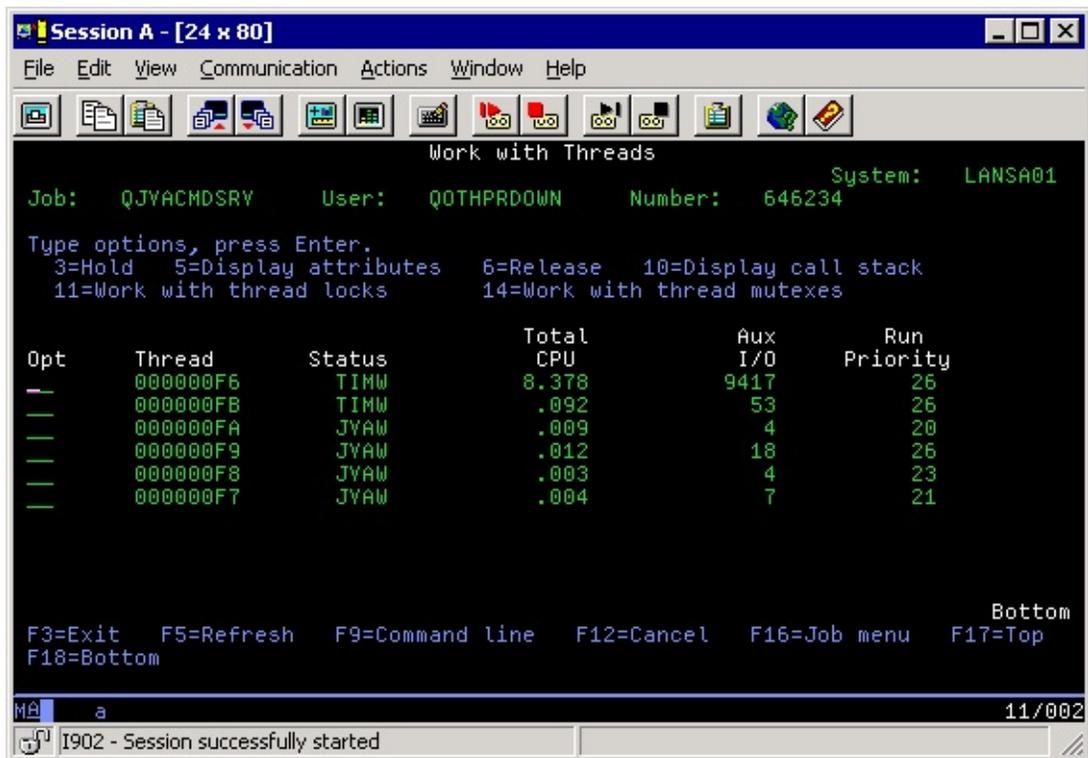
Basic system information can be obtained from the JSM console such as total and free memory and the current number of threads running in the JVM.



The active service count is the number of service threads that are running. For more information, refer to [Java Service Manager Console](#).



Use the Work with threads or the Java Virtual Machine options from the WRKACTJOB command to view Java runtime information. The following image illustrates the *Work with Threads* options.

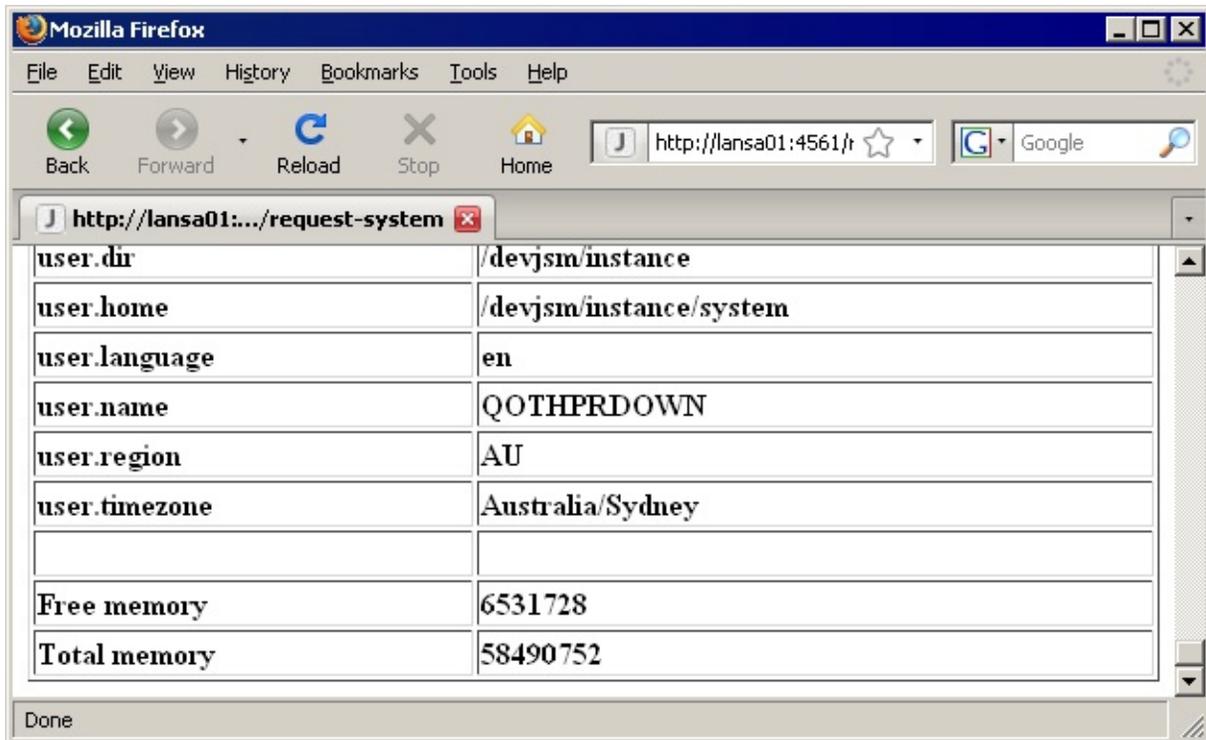


The console system information displays the JVM system properties, but also the free and total memory amounts are displayed.

When the system is running services the free memory will decrease and increase as objects are created and reclaimed by the garbage collection process.

If more memory is required the total memory will increase.

Total memory will never decrease and memory will be only returned to the operating system when the JVM environment ends.



Appendix B. Client Application Examples

This appendix contains examples of client applications that use the supplied Java Service Manager (JSM) services.

Some of the examples include references to an XML file called ORDER.XML. This file can be found in the default instance directory. In some cases, necessary records have also been added to any relevant property files. You should follow the instructions in this appendix and in the example source code to install and use the example applications.

Examples are supplied for the most common client programming languages:

[RDML Examples](#)

[RDMLX Examples](#)

[RPG Examples](#)

[C Examples](#)

RDML Examples

Open/Close Example

FTPService Example

HTTP Service Example (XML Inbound)

HTTP Service Example (XML Outbound)

SMTPMailService Example

POP3MailService Example

SMSService Example

XMLBindFileService Example

XMLParserService Example 1

XMLParserService 2 (Node traversal)

MQSeries Built-in Function Example

MQSeries and DataQueue programs Example

OpenLDAPService Example

RFIDDataSourceService Example

ZipService Example

PDFSpoolFileService Example

SVFileService Example

ExcelReadService Example

SQLService Examples

JSMDirect Example

Open/Close Example

Use this example code to test that your install has been successful.
Opens and closes the connection to the Java Service Manager.

```
***** Beginning of RDML commands *****
FUNCTION  OPTIONS(*DIRECT)
*****

DEFINE   FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE   FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
*****

***** Open service
*****

USE      BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
*****

IF      COND('#JSMSTS *NE OK')
DISPLAY  FIELDS((#JSMSTS) (#JSMMSG))
ENDIF
*****

***** Close service
*****

USE      BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
IF      COND('#JSMSTS *NE OK')
DISPLAY  FIELDS((#JSMSTS) (#JSMMSG))
ENDIF
***** End of RDML commands *****
```

FTPService Example

This function connects to a remote FTP server and allows a file to be PUT onto the remote server. The current directory on the remote server can be set as required.

Modify the default values to be displayed to suit your site.

* Uses Integrator Services: FTPSERVICE

* Loads FTPService service then connects

* to a host using FTP and puts a file in binary mode.

* Beginning of RDML commands *****

```
FUNCTION OPTIONS(*DIRECT)
```

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
```

```
DEFINE FIELD(#JSMSEVER) TYPE(*char) LENGTH(20)
```

```
DEFINE FIELD(#JSMUSER) TYPE(*CHAR) LENGTH(010) LABEL('User')
```

```
DEFINE FIELD(#JSMPSWRD) TYPE(*char) LENGTH(10)
```

```
DEFINE FIELD(#JSMDIR) TYPE(*CHAR) LENGTH(78) COLHDG('Chang
```

```
DEFINE FIELD(#JSMFROM) TYPE(*CHAR) LENGTH(78) COLHDG('Froi
```

```
DEFINE FIELD(#JSMTO) TYPE(*CHAR) LENGTH(78) COLHDG('To file')
```

* set default values for screen

```
CHANGE FIELD(#STD_INSTR) TO('Set your appropriate values then press
```

```
CHANGE FIELD(#JSMSEVER) TO(ISERIES01)
```

```
CHANGE FIELD(#JSMUSER) TO(USER)
```

```
CHANGE FIELD(#JSMPSWRD) TO(PASSWORD)
```

* Open service

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMSTS #JSMMSG)
```

```
* Load service
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVI
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
MESSAGE MSGTXT('FTPService loaded')

BEGIN_LOOP

* Format and Process JSM commands
REQUEST FIELDS((#STD_INSTR *L004 *P002 *OUTPUT *NOID) (#JSM

IF_KEY WAS(*USERKEY1)

EXECUTE SUBROUTINE(CONNECT)

* get the current directory
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GETDIR') TO_GET(#JSM
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
CHANGE FIELD(#JSMDIR) TO(#jsmmmsg)

* Quit FTP Session
EXECUTE SUBROUTINE(FTPQUIT)
CONTINUE
ENDIF

IF_KEY WAS(*MENU)

* Close service
EXECUTE SUBROUTINE(DISCONNECT)
MENU

ENDIF

* Connect to remote FTP server
EXECUTE SUBROUTINE(CONNECT)

* Transfer file
EXECUTE SUBROUTINE(TRANSFER)

END_LOOP
```

SUBROUTINE NAME(CONNECT)

* Connect to host

```
USE BUILTIN(TCONCAT) WITH_ARGS('CONNECT HOST(' #JMSERVE
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* Login

```
USE BUILTIN(TCONCAT) WITH_ARGS('LOGIN USER(' #JSMUSER ') PA
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
MESSAGE MSGTXT('Connection to remote server established')
```

ENDROUTINE

SUBROUTINE NAME(TRANSFER)

* Change directory

```
IF COND('#JSMDIR *NE *BLANK')
```

```
USE BUILTIN(TCONCAT) WITH_ARGS('CHGDIR PATH(' #JSMDIR ')') TO(
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
MESSAGE MSGTXT('Current directory has been changed on remote server')
```

ENDIF

* Binary mode

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('BINARY') TO_GET(#JSM
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* Put file

```
USE BUILTIN(TCONCAT) WITH_ARGS('PUT FROM(' #JSMFROM ') TO(
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
MESSAGE MSGTXT('File has been put on remote server')
```

* Quit after each file send
EXECUTE SUBROUTINE(FTPQUIT)

ENDROUTINE

SUBROUTINE NAME(FTPQUIT)

* Quit after each file send
USE BUILTIN(JSM_COMMAND) WITH_ARGS('QUIT') TO_GET(#JSMST
EXECUTE SUBROUTINE(CHECK) WITH_PARS(#JSMSTS #JSMMSG)
MESSAGE MSGTXT('FTP connection closed')

ENDROUTINE

SUBROUTINE NAME(DISCONNECT)

* Unload service
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TC
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)

* Close service
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARS(#JSMSTS #JSMMSG)

ENDROUTINE

* Check the JSM return status
SUBROUTINE NAME(CHECK) PARS((#JSMSTS *RECEIVED) (#JSM

IF COND('#JSMSTS *NE OK')
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMSTS ' : ' #JSMMSG) TO_GE
MENU MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#STD_TEXTL)
ENDIF

ENDROUTINE

* End of RDML commands *****

HTTP Service Example (XML Inbound)

Loads the HTTPService service. Receives xml order using Inbound xml content handler and stylesheet (created using the XML Transformation Wizard).

```
***** Beginning of RDML commands *****
FUNCTION  OPTIONS(*DIRECT)
*****

DEFINE    FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE    FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE    FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
*****

DEFINE    FIELD(#ORDER) TYPE(*CHAR) LENGTH(10)
DEFINE    FIELD(#NAME) TYPE(*CHAR) LENGTH(50)
DEFINE    FIELD(#STREET) TYPE(*CHAR) LENGTH(50)
DEFINE    FIELD(#CITY) TYPE(*CHAR) LENGTH(50)
DEFINE    FIELD(#STATE) TYPE(*CHAR) LENGTH(5)
DEFINE    FIELD(#ZIP) TYPE(*CHAR) LENGTH(5)
*****

DEFINE    FIELD(#LINENUM) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEFINE    FIELD(#PARTNUM) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEFINE    FIELD(#PARTDSC) TYPE(*CHAR) LENGTH(50)
DEFINE    FIELD(#PARTAMT) TYPE(*DEC) LENGTH(10) DECIMALS(2)
DEFINE    FIELD(#PARTQTY) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEFINE    FIELD(#FILL1) TYPE(*CHAR) LENGTH(3)
*****

DEF_LIST  NAME(#WRKLIST) FIELDS((#FILL1) (#LINENUM) (#PARTN
    (#PARTDSC) (#PARTAMT) (#PARTQTY)) TYPE(*WORKING)
*****

***** 'Open service'
*****

USE        BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE    SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****

***** 'Load service'
*****

USE        BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SE
    TPSERVICE') TO_GET(#JSMSTS #JSMMSG)
EXECUTE    SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```

*****
***** Receive XML order, SV order, POST order
*****
***** HANDLER(IXML) XSL(RECEIVEORDER) or
***** HANDLER(IXML) XSL(SENDORDER) or
***** HANDLER(ISVL) SVHEAD(ORDER) or
***** HANDLER(INVL) BIND(*FIELD)
***** HANDLER(IMP) SVHEAD(ORDER)
*****
CHANGE FIELD(#JSMCMD) TO('RECEIVE HANDLER(ISVL) SVHEA
SERVICE_LIST(FILL1,LINENUM,PARTNUM,PARTDSC,PARTAMT
PARTQTY)')
*****
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
#JSMMSG #WRKLST)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** Process and validate
*****
SELECTLIST NAMED(#WRKLST)
*****
ENDSELECT
*****
***** Send response
*****
CHANGE FIELD(#JSMCMD) TO('SEND HANDLER(ISVL) SERVICE_L
1,LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
*****
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
#JSMMSG #WRKLST)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** 'Close service'
*****
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****
***** SUB ROUTINES
*****

```

```
SUBROUTINE NAME(CHECK) PARS((#JSMSTS *RECEIVED) (#JSM  
D))
```

```
*****
```

```
IF      COND('#JSMSTS *NE OK')
```

```
*****
```

```
***** Close service and send the HTTP response
```

```
*****
```

```
USE     BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
*****
```

```
ENDIF
```

```
*****
```

```
ENDROUTINE
```

```
***** End of RDML commands *****
```

HTTP Service Example (XML Outbound)

Load HTTPService service. Creates xml order using Outbound xml content handler and stylesheet (created using the XML Transformation Wizard).

This example sends a partial order via xml to a host which returns the completed details.

The following function should be executed on the IBM i.

```
FUNCTION  OPTIONS(*DIRECT)
*****
DEFINE   FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE   FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE   FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
*****
DEFINE   FIELD(#ORDER) TYPE(*CHAR) LENGTH(010)
DEFINE   FIELD(#NAME) TYPE(*CHAR) LENGTH(050)
DEFINE   FIELD(#STREET) TYPE(*CHAR) LENGTH(050)
DEFINE   FIELD(#CITY) TYPE(*CHAR) LENGTH(050)
DEFINE   FIELD(#STATE) TYPE(*CHAR) LENGTH(005)
DEFINE   FIELD(#ZIP) TYPE(*CHAR) LENGTH(005)
*****
DEFINE   FIELD(#LINENUM) TYPE(*DEC) LENGTH(003) DECIMALS(
DEFINE   FIELD(#PARTNUM) TYPE(*DEC) LENGTH(003) DECIMALS(
DEFINE   FIELD(#PARTDSC) TYPE(*CHAR) LENGTH(050)
DEFINE   FIELD(#PARTAMT) TYPE(*DEC) LENGTH(010) DECIMALS(
      _CODE(3)
DEFINE   FIELD(#PARTQTY) TYPE(*DEC) LENGTH(003) DECIMALS(
      _CODE(Z)
*****
DEF_LIST  NAME(#WRKLST) FIELDS(#LINENUM #PARTNUM #PARTI
      T #PARTQTY) TYPE(*WORKING)
DEF_LIST  NAME(#DSPLST) FIELDS(#LINENUM #PARTNUM #PARTAI
      Y)
***** build partial order details
EXECUTE   SUBROUTINE(BUILDORD)
BEGIN_LOOP
***** Get completed order details
CLR_LIST  NAMED(#DSPLST)
SELECTLIST NAMED(#WRKLST)
```

```

ADD_ENTRY TO_LIST(#DSPLST)
ENDSELECT
*****
DISPLAY  FIELDS(#ORDER #NAME #STREET #CITY #STATE #ZIP) BE
        ST(#DSPLST)
***** 'Open service'
USE      BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** 'Load service'
USE      BUILTIN(JSM_COMMAND) WITH_ARGS("SERVICE_LOAD SE
        HTTPSERVICE") TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
***** 'Send REquest'
*****
CHANGE   FIELD(#JSMCMD) TO('SEND HANDLER(OXML) XSL(SEN
        ST(L04TST:85) URI(/CGI-BIN/JSMDIRECT?
ORDERXML) SERVICE_
        LIST(LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
*****
USE      BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
        #JSMMSG #WRKLST)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** 'Receive response'
***** HANDLER(IXML) XSL(RECEIVEORDER) or
***** HANDLER(ISVL)
*****
CHANGE   FIELD(#JSMCMD) TO('RECEIVE HANDLER(IXML) XSL(RF
        ER) SERVICE_LIST(LINENUM,PARTNUM,PARTDSC,PARTAMT,P
        Y)')
CLR_LIST NAMED(#WRKLST)
USE      BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
        #JSMMSG #WRKLST)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** 'Unload service'
*****
USE      BUILTIN(JSM_COMMAND) WITH_ARGS("SERVICE_UNLOAI

```

```

        _GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** 'Close service'
USE      BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
END_LOOP
***** SUB ROUTINES
*****
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSM
D))
IF      COND('#JSMSTS *NE OK')
DISPLAY FIELDS(#JSMSTS #JSMMSG)
*****
USE      BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****
MENU    MSGTXT('Java service error has occurred')
*****
ENDIF
ENDROUTINE
*****
SUBROUTINE NAME(BUILDORD)
***** customer details
CHANGE  FIELD(#ORDER) TO("A123456789")
CHANGE  FIELD(#NAME) TO("Uluru Vacations")
CHANGE  FIELD(#STREET) TO("1 Main Street")
CHANGE  FIELD(#CITY) TO("Alice Springs")
CHANGE  FIELD(#STATE) TO(NT)
CHANGE  FIELD(#ZIP) TO("5000")
***** create 2 order lines
CHANGE  FIELD(#PARTDSC #PARTAMT #PARTQTY) TO(*DEFAULT)
CHANGE  FIELD(#LINENUM) TO(001)
CHANGE  FIELD(#PARTNUM) TO(123)
ADD_ENTRY TO_LIST(#WRKLST)
CHANGE  FIELD(#LINENUM) TO(002)
CHANGE  FIELD(#PARTNUM) TO(456)
ADD_ENTRY TO_LIST(#WRKLST)
ENDROUTINE

```

This function processes the request and returns an updated order list. An entry in dc@w29 should specify this function e.g.

Service Name	Local Host	Process Name	Function Name
ORDERXML	*DEFAULT	JSM	EX06

FUNCTION OPTIONS(*DIRECT)

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
```

```
DEFINE FIELD(#ORDER) TYPE(*CHAR) LENGTH(010)
DEFINE FIELD(#NAME) TYPE(*CHAR) LENGTH(050)
DEFINE FIELD(#STREET) TYPE(*CHAR) LENGTH(050)
DEFINE FIELD(#CITY) TYPE(*CHAR) LENGTH(050)
DEFINE FIELD(#STATE) TYPE(*CHAR) LENGTH(005)
DEFINE FIELD(#ZIP) TYPE(*CHAR) LENGTH(005)
```

```
DEFINE FIELD(#LINENUM) TYPE(*DEC) LENGTH(003) DECIMALS(0)
DEFINE FIELD(#PARTNUM) TYPE(*DEC) LENGTH(003) DECIMALS(0)
DEFINE FIELD(#PARTDSC) TYPE(*CHAR) LENGTH(050)
DEFINE FIELD(#PARTAMT) TYPE(*DEC) LENGTH(010) DECIMALS(0)
DEFINE FIELD(#PARTQTY) TYPE(*DEC) LENGTH(003) DECIMALS(0)
DEF_LIST NAME(#WRKLIST) FIELDS(#LINENUM #PARTNUM #PARTDSC #PARTQTY) TYPE(*WORKING)
```

***** 'Open service'

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

***** 'Load service'

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS("SERVICE_LOAD SE HTTPSERVICE") TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

***** Receive XML order, SV order, POST order

```
CHANGE FIELD(#JSMCMD) TO('RECEIVE HANDLER(IXML) XSL(RECEIVE) SERVICE_LIST(LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMMSG #WRKLST)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

***** Process and validate

```
CHANGE FIELD(#PARTAMT) TO(*DEFAULT)
```

```
SELECTLIST NAMED(#WRKLST)
```

```
CHANGE FIELD(#PARTQTY) TO('#PARTQTY + 10')
```

```
CHANGE FIELD(#PARTAMT) TO('#PARTQTY * 125.25')
```

```
UPD_ENTRY IN_LIST(#WRKLST)
```

```
ENDSELECT
```

***** Send response

```
CHANGE FIELD(#JSMCMD) TO('SEND HANDLER(IXML) XSL(SEND) SERVICE_LIST(LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMMSG #WRKLST)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

***** 'Close service'

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

***** SUB ROUTINES

```
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSMMSG *OK))
```

```
IF COND('#JSMSTS *NE OK')
```

***** Close service and send the HTTP response

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

ENDIF
ENDROUTINE

SMTPMailService Example

This is a simple function to compose and send an email. This function does not support adding attachments to the email.

* Uses Integrator Services: SMTPMAILSERVICE

* Loads SMTPMailService service.

* Allows message detail to be entered

* and sends an email.

* Beginning of RDML commands *****

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)

DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#TXT) TYPE(*CHAR) LENGTH(78)

DEFINE FIELD(#TXT1) REFFLD(#TXT) COLHDG('Message')

DEFINE FIELD(#TXT2) REFFLD(#TXT) LABEL(' ')

DEFINE FIELD(#TXT3) REFFLD(#TXT) LABEL(' ')

DEFINE FIELD(#TXT4) REFFLD(#TXT) LABEL(' ')

DEF_LIST NAME(#MESSAGE) FIELDS(#TXT) TYPE(*WORKING)

DEFINE FIELD(#TOADDRESS) TYPE(*CHAR) LENGTH(50) LABEL(' T

DEFINE FIELD(#CCADDRESS) REFFLD(#TOADDRESS)

DEFINE FIELD(#C1ADDRESS) REFFLD(#TOADDRESS) LABEL(' CC')

DEFINE FIELD(#C2ADDRESS) REFFLD(#TOADDRESS) LABEL(' ')

DEFINE FIELD(#SUBJECT) REFFLD(#TOADDRESS) LABEL(' Subject')

DEF_LIST NAME(#CCLIST) FIELDS(#CCADDRESS) COUNTER(#LISTC

* Open service

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

```
* Load service
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVI
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
MESSAGE MSGTXT('SMTPMailService loaded')
```

```
BEGIN_LOOP
```

```
* Request information
```

```
REQUEST FIELDS((#TOADDRESS *L003 *P002) (#C1ADDRESS *L004 *
```

```
IF_KEY WAS(*MENU)
```

```
* Close service
```

```
EXECUTE SUBROUTINE(DISCONNECT)
```

```
MENU
```

```
ENDIF
```

```
EXECUTE SUBROUTINE(SENDEMAIL)
```

```
END_LOOP
```

```
SUBROUTINE NAME(SENDEMAIL)
```

```
* Set from and to addresses
```

```
USE BUILTIN(TCONCAT) WITH_ARGS('SET TO(' #TOADDRESS ') FROM
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
* Set CC addresses
```

```
IF COND('#C1ADDRESS *NE *BLANK')
```

```
CHANGE FIELD(#CCADDRESS) TO(#C1ADDRESS)
```

```
ADD_ENTRY TO_LIST(#CCLIST)
```

```
ENDIF
```

```
IF COND('#C2ADDRESS *NE *BLANK')
```

```
CHANGE FIELD(#CCADDRESS) TO(#C2ADDRESS)
```

```
ADD_ENTRY TO_LIST(#CCLIST)
```

```
ENDIF
```

```
IF COND('#LISTCOUNT *NE 0')
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET CC(*LIST) SERVICE  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)  
ENDIF
```

```
* Create message body
```

```
CHANGE FIELD(#TXT) TO(#TXT1)  
ADD_ENTRY TO_LIST(#MESSAGE)
```

```
CHANGE FIELD(#TXT) TO(#TXT2)  
ADD_ENTRY TO_LIST(#MESSAGE)
```

```
CHANGE FIELD(#TXT) TO(#TXT3)  
ADD_ENTRY TO_LIST(#MESSAGE)
```

```
CHANGE FIELD(#TXT) TO(#TXT4)  
ADD_ENTRY TO_LIST(#MESSAGE)
```

```
* Send mail
```

```
USE BUILTIN(TCONCAT) WITH_ARGS('SEND SUBJECT(' #SUBJECT ')'  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)  
MESSAGE MSGTXT('Email has been sent')
```

```
ENDROUTINE
```

```
SUBROUTINE NAME(DISCONNECT)
```

```
* Unload service
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TO  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
* Close service
```

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
ENDROUTINE
```

```
* Check the JSM return status
```

```
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSM
```

```
IF COND(#JSMSTS *NE OK')
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMSTS ' : ' #JSMMSG) TO_GE
MENU MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#STD_TEXTL)
ENDIF
```

```
ENDROUTINE
```

```
* End of RDML commands *****
```

POP3MailService Example

This function retrieves any mail messages for the nominated user from the mail server. For each message:

1. Any attachments are saved to the JSM instance directory
2. The body text is saved to a unique text file on the JSM instance directory.
3. Details of each email are also display to the user.
4. After processing each email is deleted.

- * Uses Integrator Services: POP3MAILSERVICE
- * Loads POP3MailService service.
- * Opens the post office and reads all messages.
- * For each message: get and display the from addresses,
- * subject and content type,
- * and save body and any attachments as files.
- * Finally deletes the message.

* Beginning of RDML commands *****

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)

DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#JSMSERVER) TYPE(*char) LENGTH(20)

DEFINE FIELD(#JSMUSER) TYPE(*CHAR) LENGTH(010) LABEL('User')

DEFINE FIELD(#JSMPSWRD) TYPE(*char) LENGTH(10)

DEFINE FIELD(#EMAILCNT) TYPE(*DEC) LENGTH(2) DECIMALS(0)

DEFINE FIELD(#EMAILNO) TYPE(*CHAR) LENGTH(2) COLHDG("")

DEFINE FIELD(#ADDRESS) TYPE(*CHAR) LENGTH(35) COLHDG('Add

DEFINE FIELD(#SUBJECT) TYPE(*CHAR) LENGTH(35) COLHDG('Subj

DEF_LIST NAME(#FROMLST) FIELDS(#ADDRESS) TYPE(*WORKING)

DEF_LIST NAME(#MESSAGES) FIELDS(#EMAILNO #SUBJECT #ADDR

* =====

* set default values for screen

CHANGE FIELD(#STD_INSTR) TO("Set your appropriate values then press

CHANGE FIELD(#JMSERVER) TO("99.99.99.99")

CHANGE FIELD(#JSMUSER) TO("user")

CHANGE FIELD(#JSMPSWRD) TO("password")

* Open service

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

* Load service

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVI

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

MESSAGE MSGTXT('POP3MailService loaded')

BEGIN_LOOP

* Request information

REQUEST FIELDS((#STD_INSTR *L003 *P002 *OUTPUT *NOID) (#JSM

IF_KEY WAS(*MENU)

* Close service

EXECUTE SUBROUTINE(DISCONNECT)

MENU

ENDIF

EXECUTE SUBROUTINE(GETMESSAGE)

END_LOOP

SUBROUTINE NAME(GETMESSAGE)

CLR_LIST NAMED(#MESSAGES)

CHANGE FIELD(#EMAILCNT) TO(0)

* Open post office

```
USE BUILTIN(TCONCAT) WITH_ARGS('OPEN SERVER(' #JMSERVER
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #JSMPSWRD ')') TC
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* Get count of messages

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET OBJECT(*MESSAG
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
USE BUILTIN(BCONCAT) WITH_ARGS(#JSMMSG 'messages in mail box'
ADD_ENTRY TO_LIST(#MESSAGES)
```

* loop through all the messages

```
BEGIN_LOOP
```

* Get message

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET OBJECT(*NEXTME
```

```
IF COND('#JSMSTS *EQ NOMAIL')
```

```
LEAVE
```

```
ENDIF
```

```
CHANGE FIELD(#EMAILCNT) TO('#EMAILCNT + 1')
```

```
SUBSTRING FIELD(#EMAILCNT) INTO_FIELD(#EMAILNO)
```

* Get subject

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET OBJECT(*SUBJECT
```

```
CHANGE FIELD(#SUBJECT) TO(#JSMMSG)
```

* Get from addresses

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET OBJECT(*FROMAI
```

```
SELECTLIST NAMED(#FROMLST)
```

```
ADD_ENTRY TO_LIST(#MESSAGES)
```

```
CHANGE FIELD(#SUBJECT) TO(*BLANK)
```

```
ENDSELECT
```

* Save text

```
USE BUILTIN(TCONCAT) WITH_ARGS('SAVE OBJECT(*TEXT) FILE(m
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
USE BUILTIN(TCONCAT) WITH_ARGS('Email' #EMAILNO ' saved to file
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#STD_TEXTL
```

* Save attachments

```
CHANGE FIELD(#JSMCMD) TO('SAVE OBJECT(*ATTACHMENTS)')
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
CHANGE FIELD(#STD_TEXTL) TO('Attachments saved')
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#STD_TEXTL
```

* read text

```
CHANGE FIELD(#JSMCMD) TO('READ OBJECT(*TEXT) SERVICE_LIS
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
SELECTLIST NAMED(#FROMLST)
ADD_ENTRY TO_LIST(#MESSAGES)
CHANGE FIELD(#SUBJECT) TO(*BLANK)
```

ENDSELECT

* Delete message

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('DELETE') TO_GET(#JSM
```

* Get next message

END_LOOP

* Close post office

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('CLOSE') TO_GET(#JSM
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

ENDROUTINE

SUBROUTINE NAME(DISCONNECT)

* Unload service

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TO  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
* Close service
```

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
ENDROUTINE
```

```
* Check the JSM return status
```

```
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSMMSG))
```

```
IF COND('#JSMSTS *NE OK')
```

```
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMSTS ' : ' #JSMMSG) TO_GET  
MENU MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#STD_TEXTL)
```

```
ENDIF
```

```
ENDROUTINE
```

```
* End of RDML commands *****
```

SMSService Example

This example demonstrates the use of the SMSService in an RDML function. It sends a predefined message to a predefined number. You may replace the message and mobile number here with your own hard coded values or replace them with fields that are populated from your database.

This example displays a screen after every JSM_COMMAND Built-in Function so that you can track the status of each step of the process. Such a technique can be useful for testing purpose. In production applications, end users would not want to see such screens.

* =====

*

* Description:

* This example demonstrates the use of the SMSService in
* an RDML function. It sends a predefined message to a
* predefined number. You may replace the message and
* mobile number here with your own hard coded values or
* replace them with fields that are populated from your
* database.

* This example displays a screen after every JSM_COMMAND
* Built-in Function so that you can track the status of
* each step of the process. Such a technique can be
* useful for testing purpose. In production applications,
* end users would not want to see such screens

*

* Disclaimer: The following material is supplied as
* sample material only. No warranty concerning the
* material or its use in any way whatsoever is
* expressed or implied.

*

* =====

FUNCTION OPTIONS(*DIRECT)

* The following locally defined fields are used to hold
* the parameters required within the JSM Built-in
* Functions.

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)

DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMSRV) TYPE(*CHAR) LENGTH(050)
```

* 'Open service'

```
CHANGE FIELD(#JSMSRV) TO('SERVER:4560')
USE BUILTIN(JSM_OPEN) WITH_ARGS(#JSMSRV) TO_GET(#JSMSTS #
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* 'Load service'

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVI
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SET ENCODING(ISO885
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* 'Send message'

```
CHANGE FIELD(#JSMCMD) TO('SEND TO(+61429999999) MSG(Hello.
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* 'Unload service'

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TC
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* 'Close service'

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* SUB ROUTINES

```
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSM
```

```
IF COND('#JSMSTS *NE OK')
DISPLAY FIELDS(#JSMSTS #JSMMSG)
```

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
MENU MSGTXT('Java service error has occurred')
```

```
ENDIF
```

```
ENDROUTINE
```

XMLBindFileService Example

Fragments Only

This RDML example of the XMLBindFileService makes use of many of the commands and keywords available within the service.

The XMLBindFileService is very tightly tied to the XML Binding Wizard that comes with LANSAs Integrator Studio. Before you start writing the code for your service application you should set up your bindings in the XML Binding Wizard first.

There are two ways that the XML Binding Wizard can handle repeating information in XML documents. One way is to label them as lists, and then use the LIST keyword on the GET and SET commands. Another way is to label them as fragments and use the FRAGMENT keyword on the GET and SET commands. When following the fragment approach, the GET and SET commands need to be placed into loops and run as many times as required for the repeating information.

This example uses Fragments only.

Following are the steps simulating how this service might be used to process some orders:

1. It uses the LIST command to determine what XML documents are in the inbound orders folder;
2. The LIST command places the orders into a working list, and using this working list we then use the READ and BIND commands to load and bind each XML document (one by one) to a service created using the XML Binding Wizard. In the READ command, a copy of the file is also archived to another folder;
3. The GET commands are next used to load the data from the XML documents into locally defined fields. Normally the data from each order would, of course, be uploaded into some existing application database, but for simplicity, this demonstration will just display the order in a screen;
4. After displaying each order, the CLOSE command is used to close the bind on the current file, and then the DELETE command to removes it from the inbound folders (remember that a copy of it is kept in the archive folder to be read by the READ command.
5. The next step is to create a response document that will be placed into

another folder. This is done using the BIND, SET, and then WRITE commands.

6. The final step is to use the CLOSE command again to close the bind associated with the creation of the response
7. Once this step is completed, the process is run through for the next order in the inbound orders folder. The program finishes once the last order is processed.

For this example to work, you will need to do the following steps:

[Step 1: XML Binding Wizard](#)

[Step 2: Create some folders and some sample XML documents](#)

[Step 3: RDML Function Code](#) or the alternative in [Step 3. RDMLX Form Code](#).

Step 1: XML Binding Wizard

Using the following XML document containing a sample of the order details, you should create a new project and create the appropriate bindings for an inbound service that will be loaded onto the server and used by the code. In the code example below, the name of the service used to process the inbound orders is called `inboundorder` so the solution name in the XML Binding Wizard should also be called `inboundorder`.

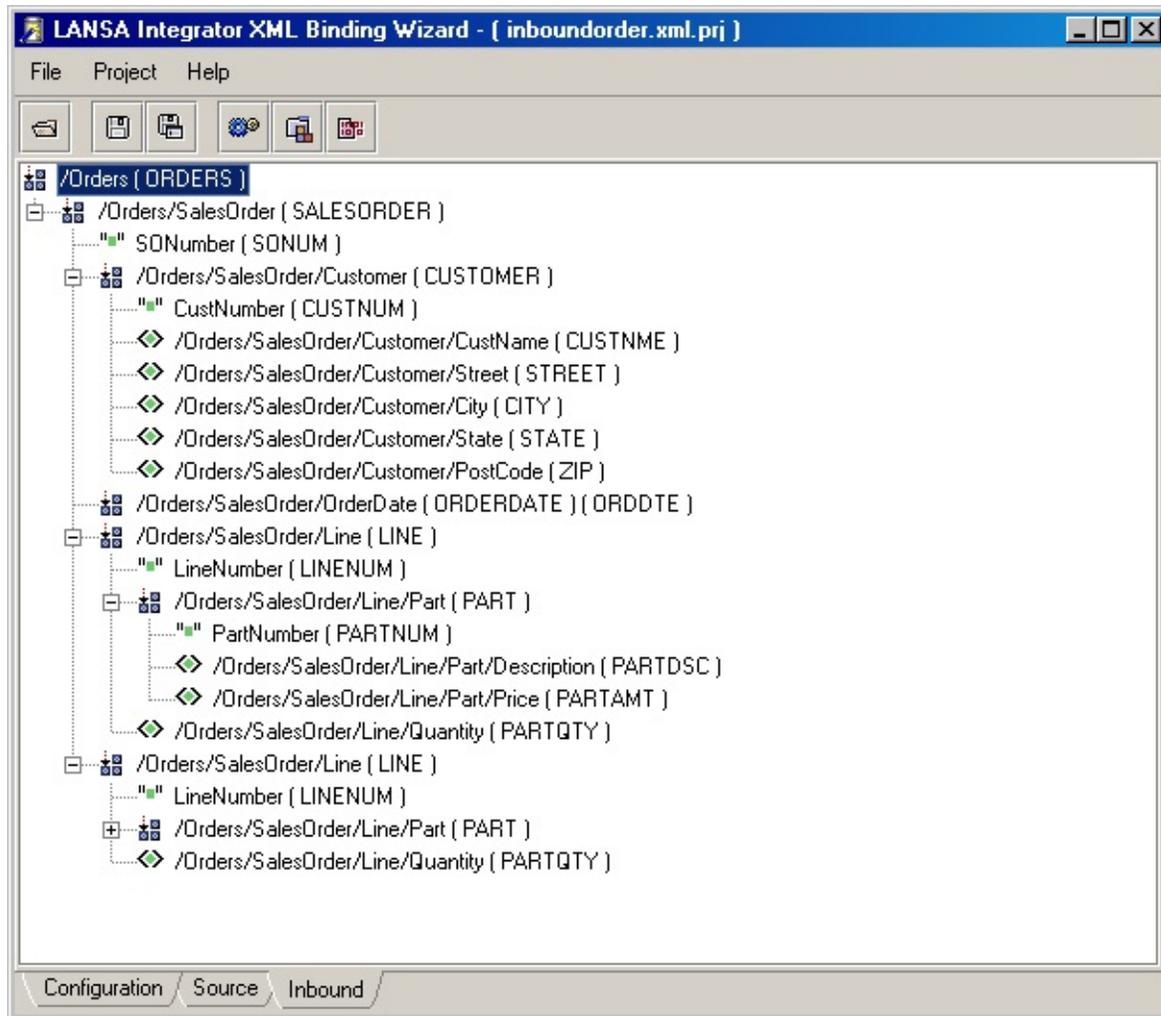
The sample XML document to use is based on the `order.xml` document that is found by default in the JSM Instance folder of your installation. The contents of the document are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<Orders>
  <SalesOrder SONumber="12345">
    <Customer CustNumber="543">
      <CustName>ABC Industries</CustName>
      <Street>123 Main St.</Street>
      <City>Chicago</City>
      <State>IL</State>
      <PostCode>60609</PostCode>
    </Customer>
    <OrderDate>981215</OrderDate>
    <Line LineNumber="1">
      <Part PartNumber="123">
        <Description>Gasket Paper</Description>
        <Price>9.95</Price>
      </Part>
      <Quantity>10</Quantity>
    </Line>
    <Line LineNumber="2">
      <Part PartNumber="456">
        <Description><![CDATA[Gasket <Polymer> Glue]]></Description>
        <Price>13.27</Price>
      </Part>
      <Quantity>5</Quantity>
    </Line>
  </SalesOrder>
</Orders>
```

</Orders>

You will need to set up your bindings as demonstrated in this screen shot.



The bindings also need to be set up for the response to the customer. The following is the sample XML document used for the response.

```
<?xml version="1.0" encoding="UTF-8"?>
<OrderResponse>
  <ResponseComments>
    <Comment>This is confirm that we have received you order</Commer
    <Comment>Your order number was 12345</Comment>
    <Comment>Your order has been successfully processed</Comment>
    <Comment>Your order details are as follow.</Comment>
  </ResponseComments>
  <SalesOrder SONumber="12345">
    <Customer CustNumber="543">
```

```
<CustName>ABC Industries</CustName>
<Street>123 Main St.</Street>
<City>Chicago</City>
<State>IL</State>
<PostCode>60609</PostCode>
</Customer>
<OrderDate>981215</OrderDate>
<Line LineNumber="1">
  <Part PartNumber="123">
    <Description>Gasket Paper</Description>
    <Price>9.95</Price>
  </Part>
  <Quantity>10</Quantity>
  <Status>Successfully processed</Status>
</Line>
<Line LineNumber="2">
  <Part PartNumber="456">
    <Description><![CDATA[Gasket <Polymer> Glue]]></Description>
    <Price>13.27</Price>
  </Part>
  <Quantity>5</Quantity>
  <Status>Unsuccessfully processed - invalid part number</Status>
</Line>
</SalesOrder>
</OrderResponse>
```

In the code example, the name of the service used to process the outbound responses is called `outboundresponse` so the solution name in the XML Binding Wizard should also be called `outboundresponse`.

The XML bindings should be mapped as follows:

LANSA Integrator XML Binding Wizard - (outboundresponse.xml.prj)

File Project Help

Icons: [Folder] [Save] [Print] [Settings] [XML] [Error]

- /OrderResponse (ORDERRESPONSE)
 - /OrderResponse/ResponseComments (RESPONSECOMMENTS)
 - /OrderResponse/ResponseComments/Comment (COMMENTS)
 - /OrderResponse/SalesOrder (SALESORDER)
 - "#" SONumber (SONUM)
 - /OrderResponse/SalesOrder/Customer (CUSTOMER)
 - "#" CustNumber (CUSTNUM)
 - /OrderResponse/SalesOrder/Customer/CustName (CUSTNME)
 - /OrderResponse/SalesOrder/Customer/Street (STREET)
 - /OrderResponse/SalesOrder/Customer/City (CITY)
 - /OrderResponse/SalesOrder/Customer/State (STATE)
 - /OrderResponse/SalesOrder/Customer/PostCode (ZIP)
 - /OrderResponse/SalesOrder/OrderDate (ORDERDATE) (ORDDTE)
 - /OrderResponse/SalesOrder/Line (LINE)
 - "#" LineNumber (LINNUM)
 - /OrderResponse/SalesOrder/Line/Part (PART)
 - "#" PartNumber (PARTNUM)
 - /OrderResponse/SalesOrder/Line/Part/Description (PARTDSC)
 - /OrderResponse/SalesOrder/Line/Part/Price (PARTAMT)
 - /OrderResponse/SalesOrder/Line/Quantity (PARTQTY)
 - /OrderResponse/SalesOrder/Line/Status (LINSTAT)
 - /OrderResponse/SalesOrder/Line (LINE)
 - "#" LineNumber (LINNUM)
 - /OrderResponse/SalesOrder/Line/Part (PART)
 - "#" PartNumber (PARTNUM)
 - /OrderResponse/SalesOrder/Line/Part/Description (PARTDSC)
 - /OrderResponse/SalesOrder/Line/Part/Price (PARTAMT)
 - /OrderResponse/SalesOrder/Line/Quantity (PARTQTY)
 - /OrderResponse/SalesOrder/Line/Status (LINSTAT)

Configuration Source Outbound

Step 2: Create some folders and some sample XML documents

1. Create the following folders immediately under the JSM Instance directory on the server.
 - neworders
 - archive
 - response
2. To simulate a number of different orders arriving in your system, create a number of XML documents based on the order.xml example and save them in the neworders folder. It pays to modify the data inside to help you differentiate between which files are being processed. (Note: it can make your testing job simpler if you save a backup of these new order files in another folder so you just need to copy them to the neworders folder each time you run a new test).

Step 3: RDML Function Code

Following is the RDML code that can be used to process these orders. Please read the notes in the code carefully as they explain what is going on. There is also an RDMLX example in [XMLBindFileService](#).

```
FUNCTION OPTIONS(*DIRECT)
```

* Define the fields to be used in this application

* Define the fields used by the JSM Commands

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
```

* Define the fields used for the orders.

* These fields will hold values read from and written to

* the XML documents

* Define the Order number and date

```
DEFINE FIELD(#SONUM) TYPE(*CHAR) LENGTH(010) LABEL('Order #
```

```
DEFINE FIELD(#ORDDTE) TYPE(*CHAR) LENGTH(010) LABEL('Order
```

* Define the Customer details

```
DEFINE FIELD(#CUSTNUM) TYPE(*CHAR) LENGTH(010) LABEL('Cust
```

```
DEFINE FIELD(#CUSTNME) TYPE(*CHAR) LENGTH(050) LABEL('Cust
```

```
DEFINE FIELD(#STREET) TYPE(*CHAR) LENGTH(050) LABEL('Street')
```

```
DEFINE FIELD(#CITY) TYPE(*CHAR) LENGTH(050) LABEL('City')
```

```
DEFINE FIELD(#STATE) TYPE(*CHAR) LENGTH(005) LABEL('State')
```

```
DEFINE FIELD(#ZIP) TYPE(*CHAR) LENGTH(005) LABEL('Post Code')
```

* Define the order line details. We will also define a

* list that holds the order line details

```
DEFINE FIELD(#LINENUM) TYPE(*DEC) LENGTH(003) DECIMALS(0)
```

```
DEFINE FIELD(#PARTNUM) TYPE(*DEC) LENGTH(003) DECIMALS(0)
```

```
DEFINE FIELD(#PARTDSC) TYPE(*CHAR) LENGTH(020) LABEL('Part I
```

```
DEFINE FIELD(#PARTAMT) TYPE(*DEC) LENGTH(010) DECIMALS(2)
```

```
DEFINE FIELD(#PARTQTY) TYPE(*DEC) LENGTH(003) DECIMALS(0)
```

```
DEFINE FIELD(#ORDTOT) TYPE(*DEC) LENGTH(010) DECIMALS(2) L
```

```
DEF_LIST NAME(#ORDLINES) FIELDS(#LINENUM #PARTNUM #PART
```

* The following field will hold the file name and path

* for the archived files

```
DEFINE FIELD(#ARCHIVE) TYPE(*CHAR) LENGTH(30) DESC('Archive
```

```
DEFINE FIELD(#X_POSN) TYPE(*DEC) LENGTH(2) DECIMALS(0) DES
```

* Define the order response details

```
DEFINE FIELD(#COMMENTS) TYPE(*CHAR) LENGTH(256) LABEL('Co
```

```
DEFINE FIELD(#LINSTAT) TYPE(*CHAR) LENGTH(20) LABEL('Line Sta
```

```
DEFINE FIELD(#RESPONSE) TYPE(*CHAR) LENGTH(30) LABEL('Resp
```

```
DEFINE FIELD(#TOGGLE) TYPE(*DEC) LENGTH(1) DECIMALS(0)
```

```
DEF_LIST NAME(#LSTCMNT) FIELDS(#COMMENTS) TYPE(*WORKIN
```

```
DEF_LIST NAME(#RSPLINES) FIELDS(#LINENUM #PARTNUM #PARTI
```

* A single field working list needs to be defined to

* hold the list of order returned from our LIST command.

* The field needs to be long enough to hold the expected

* length of the canonical file path returned from the

* LIST command.

```
DEFINE FIELD(#FILENAME) TYPE(*CHAR) LENGTH(100)
```

```
DEF_LIST NAME(#ORDERLSTW) FIELDS(#FILENAME) COUNTER(#LI
```

```
DEF_LIST NAME(#ORDERLSTB) FIELDS(#FILENAME) COUNTER(#LI
```

* We will now start the functionality by opening the JSM

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* Next we will load the JSM service - in this example we

* have selected to have tracing on.

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(XMLBINDI
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* The first thing we want to do is to get a list of all

* the orders that are in the neworders directory. To do

* this we will use the LIST command. In this scenario,

* the neworders directory is immediately under the JSM

* Instance directory on our server, so we can refer to

- * the directory simply as 'neworders'.
- * In this example the file extension has been hard coded
- * as XML. This means that only files with an extension of
- * XML will be returned.

```
CLR_LIST NAMED(#ORDERLSTW)
CHANGE FIELD(#JSMCMD) TO("LIST DIR(NEWORDERS) SERVICE_LI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

- * Display the list returned from the LIST command.

```
CLR_LIST NAMED(#ORDERLSTB)
SELECTLIST NAMED(#ORDERLSTW)
ADD_ENTRY TO_LIST(#ORDERLSTB)
ENDSELECT
DISPLAY FIELDS(#JSMCMD) BROWSELIST(#ORDERLSTB) EXIT_KEY
IF_KEY WAS(*MENU *EXIT)
```

- * Nicely close down JSM if the user has select the Exit
- * or Menu key (F3 or F12)

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
ENDIF
```

- * The next step of our application is to run through
- * the orders in our list and process the data order by
- * order. To do this we will need to use the READ, BIND,
- * and GET commands.

```
SELECTLIST NAMED(#ORDERLSTW)
```

- * Next we will use the READ command. For the READ
- * command we need to specify the file that we want
- * to access, including the file path. In this example,
- * if an order is named order1.xml for example, then the
- * FILE keyword would normally be specified as
- * FILE(neworder/order1.xml), and it would assume that the
- * neworder directory is under the JSM Instance directory
- * for the server. We could take this approach, but we
- * already have the full directory path and file name
- * specified in the list (in the field #FILENAME) so it
- * will be much simpler to use this. The actual ARCHIVE
- * file name and path will be verified in the ARCHIVE

* sub-routine.

* If we were going to be using a hard coded READ, then

* this is what it might look like:

```
* . USE BUILTIN(BUILTIN) WITH_ARGS(JSM_COMMAND)
```

```
* . WITH_ARGS('READ FILE(neworder/order1.xml)
```

```
* .     ARCHIVE(archive/arc_order1.xml)')
```

```
* . TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(ARCHIVE)
```

```
CHANGE FIELD(#JSMCMD) TO('READ FILE(')
```

```
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #FILENAME ') ARCHI
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* The next command to run is the BIND. We named the

* service to handle the inbound XML documents as

* 'inboundorder'.

```
CHANGE FIELD(#JSMCMD) TO('BIND SERVICE(INBOUNDORDER) T'
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* The next step is to retrieve the data using the GET

* command.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET FRAGMENT(ORDE
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
CHANGE FIELD(#JSMCMD) TO('GET FRAGMENT(SALESORDER) SER'
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
CHANGE FIELD(#JSMCMD) TO('GET FRAGMENT(CUSTOMER) SERVI
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
CHANGE FIELD(#JSMCMD) TO('GET FRAGMENT(ORDERDATE) SERV
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* The line details of our order technically make up a

* list. In this example though, we have defined the lines

* as fragments. As such, we need to handle them a little
* differently. We need to place the GET FRAGMENT(LINE) in
* a loop and continue in the loop until all the line
* details are retrieved. The following demonstrates how
* we do this.

```
CLR_LIST NAMED(#ORDLINES)
BEGIN_LOOP
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET FRAGMENT(LINE)
IF COND(#JSMSTS *EQ NOFRAGMENT')
LEAVE
ENDIF
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* We also need to get the part details for this line.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('GET FRAGMENT(PART)

ADD_ENTRY TO_LIST(#ORDLINES)
```

```
END_LOOP
```

* Now that we have everything for the order let us
* display it. In a real application of course you would
* carry on an load it into your ERP system or what have
* you, but we will just display it.

```
DISPLAY FIELDS(#SONUM #ORDDTE #CUSTNUM #CUSTNME #STREI
IF_KEY WAS(*MENU *EXIT)
```

* Nicely close down JSM if the user has select the Exit
* or Menu key (F3 or F12)

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
ENDIF
```

* Now that we have the data we need and have saved a copy
* of the file in the archive, we can close the bind on
* this file and delete the file from our inbound
* directory.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(CLOSE) TO_GET(#JSMSTS
CHANGE FIELD(#JSMCMD) TO('DELETE FILE(')
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #FILENAME ')') TO_
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
* Now we will work on a response XML in reply to this  
* order. To keep things clean, we will do this in a  
* separate subroutine.
```

```
EXECUTE SUBROUTINE(RESPONSE)
```

```
ENDSELECT
```

```
*
```

```
* SUB ROUTINES
```

```
* The CHECK subroutine is used to capture errors returned  
* from the JSM commands. For any errors encountered, a  
* screen will be displayed with the error message and the  
* program will exit after this.
```

```
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSM
```

```
IF COND('#JSMSTS *NE OK')
```

```
DISPLAY FIELDS(#JSMSTS #JSMMSG)
```

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
MENU MSGTXT('Java service error has occurred')
```

```
ENDIF
```

```
ENDROUTINE
```

```
* The ARCHIVE subroutine will build the file name and  
* path for the ARCHIVE keyword of the READ command.
```

```
SUBROUTINE NAME(ARCHIVE)
```

```
USE BUILTIN(SCANSTRING) WITH_ARGS(#FILENAME 'neworders' *DE  
CHANGE FIELD(#X_POSN) TO('#X_POSN + 10')
```

```
SUBSTRING FIELD(#FILENAME #X_POSN) INTO_FIELD(#ARCHIVE)
```

```
USE BUILTIN(TCONCAT) WITH_ARGS('archive/arc_' #ARCHIVE) TO_GI
```

```
ENDROUTINE
```

```
* This RESPONSE subroutine will do all the processing  
* required to build and create the response XML document.
```

SUBROUTINE NAME(RESPONSE)

- * The first thing we need to do is create a new empty
- * outbound XML document and BIND it to the outbound
- * service that we created with the XML Binding Wizard.
- * Note that we specify the type as *OUTBOUND.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('BIND SERVICE(OUTBO  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

- * Much of the information for the response we already
- * have, but a couple of fields need to be populated so
- * we will make up some fictitious data here. We will
- * create some comments, and add them to a list first.
- * The other new data we need to add is the LINSTAT field
- * to give an indication of the status of the order line.

```
CLR_LIST NAMED(#LSTCMNT)
```

```
CHANGE FIELD(#COMMENTS) TO("Thank you for your order")
```

```
ADD_ENTRY TO_LIST(#LSTCMNT)
```

```
USE BUILTIN(BCONCAT) WITH_ARGS('We have processed your order' #S
```

```
ADD_ENTRY TO_LIST(#LSTCMNT)
```

```
CHANGE FIELD(#COMMENTS) TO("Please refer below for the full details
```

```
ADD_ENTRY TO_LIST(#LSTCMNT)
```

```
CHANGE FIELD(#COMMENTS) TO("One of our sales people will be in tou
```

```
ADD_ENTRY TO_LIST(#LSTCMNT)
```

```
CHANGE FIELD(#COMMENTS) TO("For immediate assistance on your ord
```

```
ADD_ENTRY TO_LIST(#LSTCMNT)
```

```
CHANGE FIELD(#COMMENTS) TO("one more line")
```

```
ADD_ENTRY TO_LIST(#LSTCMNT)
```

```
CHANGE FIELD(#COMMENTS) TO("two more lines")
```

```
ADD_ENTRY TO_LIST(#LSTCMNT)
```

```
CHANGE FIELD(#COMMENTS) TO("three more lines")
```

```
ADD_ENTRY TO_LIST(#LSTCMNT)
```

```
CLR_LIST NAMED(#RSPLINES)
```

- * NOTE: The #TOGGLE field is used to alternate status

- * messages - to add a bit of variety.

```
SELECTLIST NAMED(#ORDLINES)
```

```

IF COND(#TOGGLE = 0)
CHANGE FIELD(#TOGGLE) TO(1)
CHANGE FIELD(#LINSTAT) TO(OK)
ELSE
CHANGE FIELD(#TOGGLE) TO(0)
CHANGE FIELD(#LINSTAT) TO('OUT OF STOCK')
ENDIF
ADD_ENTRY TO_LIST(#RSPLINES)
ENDSELECT

```

* Now that we have some data, we can start using the SET
* command to populate the outbound document object. As
* per the reading of data from the order documents, since
* this example is using fragments only, we will need to
* set up loops to add any repeating data (specifically
* the comments and the order lines).

```

CHANGE FIELD(#JSMCMD) TO('SET FRAGMENT(ORDERRESPONSE) SERVIC
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSM
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

```

```

SELECTLIST NAMED(#LSTCMNT)
CHANGE FIELD(#JSMCMD) TO('SET FRAGMENT(RESPONSECOMMENT) SERVIC
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSM
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
ENDSELECT

```

```

CHANGE FIELD(#JSMCMD) TO('SET FRAGMENT(SALESORDER) SERVICE_EXC
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSM
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

```

```

CHANGE FIELD(#JSMCMD) TO('SET FRAGMENT(CUSTOMER) SERVICE_EXC
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSM
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

```

```

SELECTLIST NAMED(#RSPLINES)
CHANGE FIELD(#JSMCMD) TO('SET FRAGMENT(LINE) SERVICE_EXC
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSM
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

```

```
CHANGE FIELD(#JSMCMD) TO('SET FRAGMENT(PART) SERVICE_EXI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMSTS #JSMMSG)
ENDSELECT
```

- * Now we have all the data prepared, we will serialize
- * the document object out to a specified file using the
- * WRITE command. We will construct a file name based on
- * the customer and order number.

```
USE BUILTIN(TCONCAT) WITH_ARGS('response/RSP_' #CUSTNUM '_' #
USE BUILTIN(TCONCAT) WITH_ARGS('WRITE FILE(' #RESPONSE ') IN
AMOUNT(1)') TO_GET(#JSMCMD)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMSTS #JSMMSG)
```

- * Finally we will CLOSE the bind then go onto the next
- * XML document.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(CLOSE) TO_GET(#JSMSTS
```

```
ENDROUTINE
```

XMLParserService Example 1

Loads XMLParserService service. Loads xml directly from a file or from an HTTP connected host or from an FTP connected host or from a POP3 post office message. Transforms the received xml to FunctionXML using a stylesheet (created using the XML Transformation Wizard) and binds to the LANSA function fields and list.

Windows Users Note: You will need to add the following line (replacing the LANSA INSTALLATION DIRECTORY with the appropriate directory) following the existing systemid.file line in the DTD Entity Resolver list in the XMLFileService.Properties, XMLParserService.Properties and XMLQueueService.Properties:

```
systemid.file:///C:/<LANSA INSTALLATION DIRECTORY>/Integrator/JSM
```

For further information, refer to [XML Entity Resolver](#).

```
***** Beginning of RDML commands *****
```

```
FUNCTION  OPTIONS(*DIRECT)
```

```
*****
```

```
DEFINE   FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
```

```
DEFINE   FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
```

```
DEFINE   FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
```

```
*****
```

```
DEFINE   FIELD(#ORDER) TYPE(*CHAR) LENGTH(10)
```

```
DEFINE   FIELD(#NAME) TYPE(*CHAR) LENGTH(50)
```

```
DEFINE   FIELD(#STREET) TYPE(*CHAR) LENGTH(50)
```

```
DEFINE   FIELD(#CITY) TYPE(*CHAR) LENGTH(50)
```

```
DEFINE   FIELD(#STATE) TYPE(*CHAR) LENGTH(5)
```

```
DEFINE   FIELD(#ZIP) TYPE(*CHAR) LENGTH(5)
```

```
*****
```

```
DEFINE   FIELD(#LINENUM) TYPE(*DEC) LENGTH(3) DECIMALS(0)
```

```
DEFINE   FIELD(#PARTNUM) TYPE(*DEC) LENGTH(3) DECIMALS(0)
```

```
DEFINE   FIELD(#PARTDSC) TYPE(*CHAR) LENGTH(50)
```

```
DEFINE   FIELD(#PARTAMT) TYPE(*DEC) LENGTH(10) DECIMALS(2)
```

```
DEFINE   FIELD(#PARTQTY) TYPE(*DEC) LENGTH(3) DECIMALS(0)
```

```
*****
```

```
DEF_LIST  NAME(#WRKLIST) FIELDS((#LINENUM) (#PARTNUM) (#PA
```

```

        #PARTAMT) (#PARTQTY)) TYPE(*WORKING)
*****
***** 'Open service'
*****
USE      BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** 'Load service'
*****
USE      BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SE
        LPARSERSERVICE') TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** Load XML from required source.
***** Select required CHANGE command according to XML source.
*****
***** Load directly from a file
CHANGE  FIELD(#JSMCMD) TO('LOAD METHOD(*FILE) FILE(order.x
*****
***** OR Load from an HTTP source
CHANGE  FIELD(#JSMCMD) TO('LOAD METHOD(*HTTP) HOST(LOC
        ) FILE(/order.xml)')
*****
***** OR Load from an FTP source
CHANGE  FIELD(#JSMCMD) TO('LOAD METHOD(*FTP) HOST(LOCA
        LE(/jsm/instance/order.xml) USER(user) PASSWORD(passwor
        d)')
*****
***** OR Load from a POP3 mail source
CHANGE  FIELD(#JSMCMD) TO('LOAD METHOD(*POP3) SERVER(1
        2) USER(testuser) PASSWORD(testuser)')
*****
USE      BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
        #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** 'Transform to FunctionXML and bind to function
*****
CHANGE  FIELD(#JSMCMD) TO('TRANSFORM XSL(ORDER) SERVIC
        NENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
*****

```

```

USE      BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
        #JSMMSG #WRKLST)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****
***** Display XML order data
*****
DISPLAY  FIELDS((#ORDER) (#NAME) (#STREET) (#CITY) (#STATE) (
        P))
*****
SELECTLIST NAMED(#WRKLST)
*****
DISPLAY  FIELDS((#LINENUM) (#PARTNUM) (#PARTDSC) (#PARTAM
        RTQTY))
ENDSELECT
***** 'Unload service'
*****
USE      BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD
        (#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****
***** 'Close service'
*****
USE      BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
*****
***** SUB ROUTINES
*****
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSM
        D))
IF      COND('#JSMSTS *NE OK')
DISPLAY  FIELDS((#JSMSTS) (#JSMMSG))
*****
USE      BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****
MENU     MSGTXT('Java service error has occurred')
*****
ENDIF
*****
ENDROUTINE

```

***** End of RDML commands *****

XMLParserService 2 (Node traversal)

This is a more complex XMLParserService example that demonstrates the node traversal of received xml. The direct binding of received xml to LANSAs fields and lists obviates the need to perform node traversal in most situations (refer to HTTPService service). This example parses xml and allows traversal of the nodes in the DOM tree to load node attribute values.

```
***** Beginning of RDML commands *****
FUNCTION  OPTIONS(*DIRECT)
*****
DEFINE   FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE   FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE   FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
*****
DEFINE   FIELD(#MESSAGE) TYPE(*CHAR) LENGTH(255)
DEF_LIST NAME(#MSGLST) FIELDS((#MESSAGE)) TYPE(*WORKING)
*****
***** 'Open service'
*****
USE      BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** 'Load service'
*****
USE      BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SE
      LPARSERSERVICE') TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** 'Load XML source'
*****
***** Load from a file
CHANGE   FIELD(#JSMCMD) TO('LOAD METHOD(*FILE) FILE(order.x
USE      BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(
      #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** 'Parse XML'
```

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('PARSE') TO_GET(#J
      #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG
```

***** 'Get ROOT tag'

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('GET OBJECT(*ROO
      )') TO_GET(#JSMSTS #JSMMSG)
```

```
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
```

***** Get warning and error messages

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('GET OBJECT(*MES:
      ERVICE_LIST(MESSAGE)) TO_GET(#JSMSTS #JSMMSG #MSGL:
```

```
SELECTLIST NAMED(#MSGLST)
```

```
DISPLAY FIELDS((#MESSAGE))
```

```
ENDSELECT
```

***** 'Get node value'

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('GET NODE(/SALES(
      STOMER/CUSTNAME)') TO_GET(#JSMSTS #JSMMSG )
```

```
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
```

***** 'Collect SALESORDER/LINE elements'

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('FOREACH NODE(/S
      R/LINE)') TO_GET(#JSMSTS #JSMMSG)
```

```
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
```

Part 2

***** 'Loop through Line elements'

```
BEGIN_LOOP
```

```
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('NEXT OBJECT(*NO
      _GET(#JSMSTS #JSMMSG)
```

```

*****
IF      COND('#JSMSTS *NE OK')
GOTO   LABEL(BRK)
ENDIF
*****
USE     BUILTIN(JSM_COMMAND) WITH_ARGS('GET NODE(*CURRENT
      ATTRIBUTE(LineNumber)') TO_GET(#JSMSTS #JSMMSG)
*****
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
*****
CHANGE  FIELD(#JSMCMD) TO('GET NODE(PART) ATTRIBUTE(PART
      )
USE     BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
      #JSMMSG)
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
*****
USE     BUILTIN(JSM_COMMAND) WITH_ARGS('GET NODE(PART/DI
      ON)') TO_GET(#JSMSTS #JSMMSG)
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
*****
END_LOOP
BRK ***** 'Loop breakout
*****
***** 'Unload service'
*****
USE     BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD
      (#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** 'Close service'
*****
USE     BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** SUB ROUTINES
*****
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSMMSG
      D))
*****

```

```
IF      COND('#JSMSTS *NE OK')
DISPLAY  FIELDS((#JSMSTS) (#JSMMSG))
*****

USE      BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****

MENU     MSGTXT('Java service error has occurred')
*****

ENDIF
*****

ENDROUTINE
```

XMLQueryService Example

```
function options(*DIRECT)

define field(#ORDERXML) type(*CHAR) length(256) desc('Path to salesorde
def_list name(#BROWSE) fields((#ORDERNUM *OUT) (#CUSTNUM *OU'

change field(#ORDERXML) to('/LANSA_dc@pgmlib/jsm/instance/order.xml
begin_loop

request fields(#ORDERXML) identify(*DESC) browselist(#browse)
clr_list named(#BROWSE)
execute subroutine(LOADORDERS)
selectlist named(#ORDERS)
add_entry to_list(#BROWSE)
endselect

end_loop

subroutine name(LOADORDERS)

define field(#ORDERNUM) type(*CHAR) length(10) colhdg('Order' 'Number
define field(#CUSTNUM) type(*CHAR) length(10) colhdg('Customer' 'Numb
def_list name(#ORDERS) fields(#ORDERNUM #CUSTNUM) type(*WORKI

* open JSM connection
* load the XMLQueryService
use builtin(JSM_OPEN) to_get(#JSMSTS #JSMMSG)
execute subroutine(CHECK) with_parms(#JSMSTS #JSMMSG)

change field(#JSMCMD) to('"service_load service(XMLQueryService) trace(*
use builtin(JSM_COMMAND) with_args(#JSMCMD) to_get(#JSMSTS #JSM
execute subroutine(CHECK) with_parms(#JSMSTS #JSMMSG)

* load the salesorder.xml file
use builtin(TCONCAT) with_args('load file(' #ORDERXML ')') to_get(#JSMC
use builtin(JSM_COMMAND) with_args(#JSMCMD) to_get(#JSMSTS #JSM
execute subroutine(CHECK) with_parms(#JSMSTS #JSMMSG)
```

```
* query each <SalesOrder> element, for each, return
* - value of @SONumber attribute (sales order number)
* - value of @CustNumber attribute of the contained <Customer> element
clr_list named(#ORDERS)
change field(#JSMCMD) to("query nodes(//SalesOrder) nodesvalue1(@SONu
use builtin(JSM_COMMAND) with_args(#JSMCMD) to_get(#JSMSTS #JSM
execute subroutine(CHECK) with_parms(#JSMSTS #JSMMSG)
```

```
* unload the XMLQueryService
* close the JSM connection
change field(#JSMCMD) to('service_unload')
use builtin(JSM_COMMAND) with_args(#JSMCMD) to_get(#JSMSTS #JSM
use builtin(JSM_CLOSE) to_get(#JSMSTS #JSMMSG)
```

```
endroutine
```

```
* -----
* Subroutine CHECK: checks the JSM return status
* -----
subroutine name(CHECK) parms((#JSMSTS *RECEIVED) (#JSMMSG *REC
```

```
if cond('#jstmsts *ne ok')
```

```
use builtin(TCONCAT) with_args(#JSMSTS ' : ' #JSMMSG) to_get(#STD_TE
menu msgid(DCM9899) msgf(dc@m01) msgdta(#STD_TEXTL)
```

```
endif
```

```
endroutine
```

MQSeries Built-in Function Example

***** Beginning of RDML commands *****

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMCNT) TYPE(*CHAR) LENGTH(4)

DEFINE FIELD(#JSMGET) TYPE(*CHAR) LENGTH(4)

DEFINE FIELD(#JSMPUT) TYPE(*CHAR) LENGTH(4)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)

DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#QMANAGER) TYPE(*CHAR) LENGTH(48)

DEFINE FIELD(#QNAME) TYPE(*CHAR) LENGTH(48)

DEFINE FIELD(#QMODE) TYPE(*CHAR) LENGTH(10)

DEFINE FIELD(#WAIT) TYPE(*DEC) LENGTH(9) DECIMALS(0)

DEFINE FIELD(#LENGTH) TYPE(*DEC) LENGTH(9) DECIMALS(0)

DEFINE FIELD(#GETMSGID) TYPE(*CHAR) LENGTH(24)

DEFINE FIELD(#GETCORID) TYPE(*CHAR) LENGTH(24)

DEFINE FIELD(#RQMANAGER) TYPE(*CHAR) LENGTH(48)

DEFINE FIELD(#RQNAME) TYPE(*CHAR) LENGTH(48)

DEFINE FIELD(#JSMMSGID) TYPE(*CHAR) LENGTH(24) DECIMALS

DEFINE FIELD(#JSMCORID) TYPE(*CHAR) LENGTH(24) DECIMALS

DEFINE FIELD(#MSGTYPE) TYPE(*DEC) LENGTH(9) DECIMALS(0)

DEFINE FIELD(#MSGFMT) TYPE(*CHAR) LENGTH(8)

DEFINE FIELD(#DEPTH) TYPE(*DEC) LENGTH(9) DECIMALS(0)

DEFINE FIELD(#CCSID) TYPE(*DEC) LENGTH(9) DECIMALS(0)

DEFINE FIELD(#PRIORITY) TYPE(*DEC) LENGTH(9) DECIMALS(0)

DEFINE FIELD(#EXPIRY) TYPE(*DEC) LENGTH(9) DECIMALS(0)

DEFINE FIELD(#PERSIST) TYPE(*CHAR) LENGTH(1)

DEFINE FIELD(#QUSER) TYPE(*CHAR) LENGTH(12)

DEFINE FIELD(#APPLID) TYPE(*CHAR) LENGTH(32)

***** Connect to queue manager

CHANGE FIELD(#QMANAGER) TO('USERAGENT.QUEUE.MANAGE.
USE BUILTIN(MQ_CONN) WITH_ARGS(#QMANAGER) TO_GET(#J
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))

***** Open GET queue

```
CHANGE FIELD(#QNAME) TO('USERAGENT.QUEUE')
USE BUILTIN(MQ_OPEN) WITH_ARGS(#JSMCNT #QNAME 'IQ') TO
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
```

***** Open PUT queue

```
USE BUILTIN(MQ_OPEN) WITH_ARGS(#JSMCNT #QNAME 'OFU')
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
```

***** Get GET queue depth

```
USE BUILTIN(MQ_DEPTH) WITH_ARGS(#JSMCNT #JSMGET) TO_
DISPLAY FIELDS((#JSMSTS) (#DEPTH))
```

***** Open JSM connection

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SE
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
```

***** Create XML message

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SEND XSL(SENDOF
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
```

***** Put XML message

```
CHANGE FIELD(#CCSID) TO(37)
CHANGE FIELD(#MSGFMT) TO(MQSTR)
CHANGE FIELD(#MSGTYPE) TO(999999999)
CHANGE FIELD(#PERSIST) TO('Y')
CHANGE FIELD(#PRIORITY) TO(-1)
CHANGE FIELD(#EXPIRY) TO(-1)
CHANGE FIELD(#EXPIRY) TO(-1)
```

```

CHANGE FIELD(#JSMMSGID) TO('MYMSG')
CHANGE FIELD(#JSMCORID) TO('MYCOR')
CHANGE FIELD(#RQMANAGER) TO('MYQMANAGER')
CHANGE FIELD(#RQNAME) TO('MYQUEUE')
CHANGE FIELD(#QUSER) TO('MYSELF')
CHANGE FIELD(#APPLID) TO('MYAPPLID')
*****

USE BUILTIN(MQ_PUT) WITH_ARGS(#JSMCNT #JSMPUT #MSGTY
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
*****

***** Get XML message
*****

CHANGE FIELD(#WAIT) TO(35)
CHANGE FIELD(#MSGTYPE) TO(0)
CHANGE FIELD(#LENGTH) TO(100000)
CHANGE FIELD(#JSMMSGID) TO(*BLANK)
CHANGE FIELD(#JSMCORID) TO(*BLANK)
CHANGE FIELD(#MSGFMT) TO(*BLANK)
CHANGE FIELD(#MSGTYPE) TO(0)
CHANGE FIELD(#RQMANAGER) TO(*BLANK)
CHANGE FIELD(#RQNAME) TO(*BLANK)
CHANGE FIELD(#QUSER) TO(*BLANK)
CHANGE FIELD(#APPLID) TO(*BLANK)
*****

USE BUILTIN(MQ_GET) WITH_ARGS(#JSMCNT #JSMGET #WAIT #
DISPLAY FIELDS((#JSMSTS) (#MSGTYPE) (#MSGFMT) (#RQMANAG
*****

***** Close JSM connection
*****

USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
*****

***** Close GET queue
*****

USE BUILTIN(MQ_CLOSE) WITH_ARGS(#JSMCNT #JSMGET) TO_
DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
*****

***** Close PUT queue
*****

```

```
USE      BUILTIN(MQ_CLOSE) WITH_ARGS(#JSMCNT #JSMPUT) TO_(  
DISPLAY  FIELDS((#JSMSTS) (#JSMMSG))
```

```
*****
```

```
***** Disconnect from queue manager
```

```
*****
```

```
USE      BUILTIN(MQ_DISC) WITH_ARGS(#JSMCNT) TO_GET(#JSMST  
DISPLAY  FIELDS((#JSMSTS) (#JSMMSG))
```

```
***** End of RDML commands *****
```

MQSeries and DataQueue programs Example

This example illustrates both MQSeries and DataQueue access in the one program.

```
***** Beginning of RDML commands *****
FUNCTION  OPTIONS(*DIRECT)
*****

DEFINE   FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE   FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE   FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
*****

***** Stop MQ fields going across by prefixing with JSM
*****

DEFINE   FIELD(#JSMQMGR) TYPE(*CHAR) LENGTH(48)
DEFINE   FIELD(#JSMQNAME) TYPE(*CHAR) LENGTH(48)
*****

DEFINE   FIELD(#JSMRQMGR) TYPE(*CHAR) LENGTH(48)
DEFINE   FIELD(#JSMREPLYQ) TYPE(*CHAR) LENGTH(48)
*****

DEFINE   FIELD(#JSMWAIT) TYPE(*DEC) LENGTH(8) DECIMALS(0)
DEFINE   FIELD(#JSMMSIZE) TYPE(*DEC) LENGTH(8) DECIMALS(0)
DEFINE   FIELD(#JSMDEPTH) TYPE(*DEC) LENGTH(8) DECIMALS(0)
DEFINE   FIELD(#JSMCCSID) TYPE(*DEC) LENGTH(8) DECIMALS(0)
DEFINE   FIELD(#JSMPOST) TYPE(*CHAR) LENGTH(1)
DEFINE   FIELD(#JSMPTYP) TYPE(*DEC) LENGTH(8) DECIMALS(0)
DEFINE   FIELD(#JSMID) TYPE(*CHAR) LENGTH(24)
DEFINE   FIELD(#JSMCID) TYPE(*CHAR) LENGTH(24)
DEFINE   FIELD(#JSMMTYPE) TYPE(*DEC) LENGTH(9) DECIMALS(0)
*****

***** Data Queue
*****

DEFINE   FIELD(#JSMQLIB) TYPE(*CHAR) LENGTH(10)
DEFINE   FIELD(#JSMQNAME) TYPE(*CHAR) LENGTH(10)
DEFINE   FIELD(#JSMQKEY) TYPE(*CHAR) LENGTH(256)
*****

DEFINE   FIELD(#ORDER) TYPE(*CHAR) LENGTH(10)
DEFINE   FIELD(#NAME) TYPE(*CHAR) LENGTH(50)
DEFINE   FIELD(#STREET) TYPE(*CHAR) LENGTH(50)
```

```

DEFINE FIELD(#CITY) TYPE(*CHAR) LENGTH(50)
DEFINE FIELD(#STATE) TYPE(*CHAR) LENGTH(5)
DEFINE FIELD(#ZIP) TYPE(*CHAR) LENGTH(5)
*****

DEFINE FIELD(#LINENUM) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEFINE FIELD(#PARTNUM) TYPE(*DEC) LENGTH(3) DECIMALS(0)
DEFINE FIELD(#PARTDSC) TYPE(*CHAR) LENGTH(50)
DEFINE FIELD(#PARTAMT) TYPE(*DEC) LENGTH(10) DECIMALS(2)
DEFINE FIELD(#PARTQTY) TYPE(*DEC) LENGTH(3) DECIMALS(0)
*****

*****

DEF_LIST NAME(#WRKLIST) FIELDS((#LINENUM) (#PARTNUM) (#PARTAMT) (#PARTQTY)) TYPE(*WORKING)
*****

CHANGE FIELD(#JSMQMGR) TO('TEST.QMANAGER')
CHANGE FIELD(#JSMQNAME) TO('TEST.QUEUE')
*****

***** Get Queue Depth
*****

CALL PGM(JSMMQDEPTH) PARM(#JSMQMGR #JSMQNAME #JSMNUM_LEN (*DEFINED)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****

DISPLAY FIELDS((#JSMDEPTH))
*****

***** Open service
*****

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****

***** Load service
*****

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SETO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****

***** Get message
*****

CHANGE FIELD(#JSMQLIB) TO(JSMSRC)

```

```

CHANGE  FIELD(#JSMQNME) TO(JSMQUEUE)
CHANGE  FIELD(#JSMQKEY) TO(*BLANK)
CHANGE  FIELD(#JSMWAIT) TO(5000)
*****
CALL    PGM(JSMDQGET) PARM(#JSMQLIB #JSMQNME #JSMQKEY
      #JSMSTS #JSMMSG) NUM_LEN(*DEFINED
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
CALL    PGM(JSMMQGET) PARM(#JSMQMGR #JSMQNAME #JSMWA
      #JSM MID #JSMCID #JSMRQMGR #JSMREPLYQ #JSM MTYPE #JSM
      NUM_LEN(*DEFINED)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** Receive XML
*****
CHANGE  FIELD(#JSMCMD) TO('RECEIVE XSL(RECEIVEORDER) SE
      (LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY)')
*****
USE     BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
      (#JSMSTS #JSMMSG #WRKLST)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
*****
DISPLAY FIELDS((#ORDER))
*****
***** Send XML
*****
CHANGE  FIELD(#JSMCMD) TO('SEND XSL(SENDORDER)
      SERVICE_LIST(LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTC
*****
USE     BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD)
      TO_GET(#JSMSTS #JSMMSG #WRKLST)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
*****
***** Put new message
*****
CHANGE  FIELD(#JSMCCSID) TO(0)
CHANGE  FIELD(#JSM MTYPE) TO(8)
CHANGE  FIELD(#JSM MID) TO(*BLANK)

```

```

CHANGE FIELD(#JSMCID) TO(*BLANK)
CHANGE FIELD(#JSMPST) TO(Y)
CHANGE FIELD(#JSMPTY) TO(4)
CHANGE FIELD(#JSMRQMGR) TO(*BLANK)
CHANGE FIELD(#JSMREPLYQ) TO(*BLANK)
*****

CALL PGM(JSMMQPUT) PARM(#JSMQMGR #JSMQNAME #JSMCC:
    #JSMPTY #JSM MID #JSMCID #JSM MTTYPE #JSMRQMGR #JSMRE
    #JSMSTS #JSMMSG) NUM_LEN(*DEFINED)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****

CALL PGM(JSMDQPUT) PARM(#JSMQLIB #JSMQNAME #JSMQKEY
    #JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****

***** Close service
*****

USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSC
*****

***** SUB ROUTINES
*****

SUBROUTINE NAME(CHECK) PARM((#JSMSTS *RECEIVED) (#JSM
*****

IF COND('#JSMSTS *NE OK')
*****

DISPLAY FIELDS((#JSMSTS) (#JSMMSG))
*****

USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
*****

MENU MSGTXT('Java service error has occurred')
*****

ENDIF
*****

ENDROUTINE
***** End of RDML commands *****

```

OpenLDAPService Example

This example demonstrates the use of the OpenLDAPService in an RDML function. It requests the user to provide a user name (a 'cn' value) and then GETs a record from the IBM Directory Server (IBMTelDir) on the IBM i. Two screens are returned. The first one shows what is returned when you use the attribute *ALL, while the second shows you how to return only selected attributes.

To run this example you will need to do the following:

1. Configure and start the IBM Directory Server on the IBM i. For more information on the IBM Directory Server please refer to the relevant IBM Manuals
2. Define two fields in the LANSAs Repository:
 - #JSMSTS, Char, Length 20
 - #JSMMSG, Char, Length 255
3. Set the #JSMSRV field to the correct JSM server value
4. Set the #LDPSRV field to the correct LDAP server value

* =====

* Description:

* This example demonstrates the use of the
* OpenLDAPService in an RDML function. It requests the
* user to provide a user name (a 'cn' value) and then
* GETs a record from the IBM Directory Server (IBMTelDir)
* on the IBM i . Two screens are returned. The first one
* shows what is returned when you use the attribute *ALL,
* while the second shows you how to return only selected
* attributes.

* To run this example you will need to do the following:

* Step 1.

* Configure and start the IBM Directory Server on the
* IBM i. For more information on the IBM Directory
* Server please refer to the relevant IBM Manuals

* Step 2

* Define two fields in the LANSAs Repository:

* - #JSMSTS, Char, Length 20

- * - #JSMMSG, Char, Length 255
- * Step 3
- * Set the #JSMSRV field to the correct JSM server value
- * Step 4
- * Set the #LDPSRV field to the correct LDAP server value

- * Disclaimer: The following material is supplied as
- * sample material only. No warranty concerning the
- * material or its use in any way whatsoever is
- * expressed or implied.

* =====
 FUNCTION OPTIONS(*DIRECT)

- * The following locally defined fields are used to hold
- * the parameters required within the JSM Built-in
- * functions.

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMSRV) TYPE(*CHAR) LENGTH(050)
DEFINE FIELD(#LDPSRV) TYPE(*CHAR) LENGTH(050)
```

- * The following fields are used in the working list
- * required for the GET commands.

```
DEFINE FIELD(#ATNAME) TYPE(*CHAR) LENGTH(035)
DEFINE FIELD(#ATVALUE) TYPE(*CHAR) LENGTH(035)
DEF_LIST NAME(#WRKLST) FIELDS(#ATNAME #ATVALUE) TYPE(*W
DEF_LIST NAME(#GET_LIST) FIELDS(#ATNAME #ATVALUE)
DEF_LIST NAME(#GET_LIST2) FIELDS(#ATNAME #ATVALUE)
```

- * Define the field used to capture the 'cn' value (that
- * is, the user name).

```
DEFINE FIELD(#CN_FNAME) TYPE(*CHAR) LENGTH(050) DESC('Nam
```

- * 'Open service'
- * The JSM_OPEN Builtin Function is used to connect this
- * JSM client to the Java Services Manager, and to start
- * a thread for the service.

```
CHANGE FIELD(#JSMSRV) TO('<system-name>:<port>')
```

```
USE BUILTIN(JSM_OPEN) WITH_ARGS(#JSMSTVS #
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTVS #JSMMSG)
```

* 'Load service'

- * The Service_Load(OpenLDAPService) command loads and
- * initializes the service.

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(OPENLDAI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTVS #JSMMSG)
```

* 'Bind to LDAP Server'

- * The BIND command is used to establish a connection to
- * the LDAP server. In this scenario we are passing an
- * authentication name (DN) and password, but these are
- * not mandatory for this command

```
CHANGE FIELD(#LDPSRV) TO('<LDAP-server-name>')
CHANGE FIELD(#JSMCMD) TO('"BIND HOST(")
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #LDPSRV ') DN(cn=A
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTVS #JSMMSG)
```

- * Execute a subroutine to REQUEST the User details to be
- * retrieved, then run the GET command. Keep looping until
- * user Cancels or Exits. Notice that the Exit and Menu
- * keys are controlled to ensure that the UNLOAD and CLOSE
- * commands are always executed.

```
DUNTIL COND('(#IO$KEY *EQ "12") *OR (#IO$KEY *EQ "03"))'
EXECUTE SUBROUTINE(REQ_USER)
ENDUNTIL
```

```
CHANGE FIELD(#JSMCMD) TO(UNBIND)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTVS #JSMMSG)
```

* 'Unload service'

- * This command is required to unload the service and to
- * remove the temporary directory. If you needed to send
- * out multiple messages then you would not issue this
- * command until after you had finished sending all the

* messages.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TC
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* 'Close service'

* The final step in the process is to close the service.

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* SUBROUTINES

* This subroutine asks the user to provide the CN name

* they wish to retrieve

```
SUBROUTINE NAME(REQ_USER)
```

```
REQUEST FIELDS(#CN_FNAME) EXIT_KEY(*YES *RETURN) MENU_I
```

* 'GET Command'

* This command is used to get the record of with the user

* that has been selected.

* Two screens are returned. The first one shows what is

* returned when you use the attribute *ALL, while the

* second shows you how to return only selected attributes

* Populate the first screen using ATTRIBUTES(*ALL)

```
CLR_LIST NAMED(#WRKLST)
```

```
CHANGE FIELD(#JSMCMD) TO('GET DN(cn=')
```

```
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #CN_FNAME ' ', cn=us
```

```
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD ' ATTRIBUTES(*ALL
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
CLR_LIST NAMED(#GET_LIST)
```

```
SELECTLIST NAMED(#WRKLST)
```

```
ADD_ENTRY TO_LIST(#GET_LIST)
```

```
ENDSELECT
```

```
DISPLAY BROWSELIST(#GET_LIST) EXIT_KEY(*YES *RETURN) MEN
```

* Populate the second screen using ATTRIBUTES(cn,sn,mail)

```
CLR_LIST NAMED(#WRKLST)
CHANGE FIELD(#JSMCMD) TO('GET DN(cn=')
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #CN_FNAME ', cn=u:
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD ' ATTRIBUTES(cn,sn,
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
CLR_LIST NAMED(#GET_LIST2)
SELECTLIST NAMED(#WRKLST)
ADD_ENTRY TO_LIST(#GET_LIST2)
ENDSELECT
```

```
DISPLAY BROWSELIST(#GET_LIST2) EXIT_KEY(*YES *RETURN) ME
ENDROUTINE
```

```
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSM
```

```
IF COND('#JSMSTS *NE OK')
DISPLAY FIELDS(#JSMSTS #JSMMSG)
```

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
MENU MSGTXT('Java service error has occurred')
```

```
ENDIF
```

```
ENDROUTINE
```

RFIDataSourceService Example

This LANSA function is the host program for the client RFIExample example program.

```
***** Beginning of RDML commands *****
FUNCTION  OPTIONS(*DIRECT)
DEFINE   FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE   FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE   FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
DEFINE   FIELD(#EMPNO) TYPE(*CHAR) LENGTH(5)
DEFINE   FIELD(#SURNAME) TYPE(*CHAR) LENGTH(20)
DEFINE   FIELD(#GIVENAME) TYPE(*CHAR) LENGTH(20)
DEFINE   FIELD(#ADDRESS1) TYPE(*CHAR) LENGTH(25)
DEFINE   FIELD(#ADDRESS2) TYPE(*CHAR) LENGTH(25)
DEFINE   FIELD(#ADDRESS3) TYPE(*CHAR) LENGTH(25)
DEFINE   FIELD(#SKILCODE) TYPE(*CHAR) LENGTH(10)
DEFINE   FIELD(#DATEACQR) TYPE(*DEC) LENGTH(6) DECIMALS(0)
DEFINE   FIELD(#GRADE) TYPE(*CHAR) LENGTH(1)
DEFINE   FIELD(#COMMENT) TYPE(*CHAR) LENGTH(20)
DEF_LIST NAME(#SKILL) FIELDS((#SKILCODE) (#DATEACQR) (#GR
      COMMENT)) TYPE(*WORKING)
DEFINE   FIELD(#SNAME) TYPE(*CHAR) LENGTH(10)
DEFINE   FIELD(#LNAME) TYPE(*CHAR) LENGTH(30)
DEF_LIST NAME(#MAPLST) FIELDS((#SNAME) (#LNAME)) TYPE(*WORKING)
DEF_LIST NAME(#FLDLST) FIELDS((#SNAME)) TYPE(*WORKING)
DEFINE   FIELD(#SOURCE) TYPE(*CHAR) LENGTH(50)
***** Build Map
EXECUTE  SUBROUTINE(BLDMAP)
***** 'Open service'
USE      BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
***** 'Load service'
CHANGE   FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(RFIDATA
      SERVICE_LIST(SNAME,LNAME) BIND(*FIELD))')
USE      BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
      (#JSMMSG #MAPLST)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
***** Check for inbound data source
```

```

USE    BUILTIN(JSM_COMMAND) WITH_ARGS('CHECK OBJECT(*D
      TO_GET(#JSMSTS #JSMMSG)
IF    COND('#JSMSTS *EQ OK')
***** Get employee id from data source
CHANGE  FIELD(#JSMCMD) TO('GET OBJECT(*FIELD) NAME(EMPNO
      CE_LIST(SNAME)')
USE    BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
      #JSMMSG #FLDLST)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
CHANGE  FIELD(#JSMCMD) TO('GET OBJECT(*TABLE) NAME(SKILL
      ICE_LIST(SKILCODE,DATEACQR,GRADE,COMMENT)')
USE    BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
      #JSMMSG #SKILL)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
ENDIF
***** Create data source
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('CREATE DATASOURC
      E') TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
***** Add skills to data source
EXECUTE  SUBROUTINE(SKILL)
CHANGE  FIELD(#JSMCMD) TO('PUT OBJECT(*TABLE) NAME(SKILL
      ICE_LIST(SKILCODE,DATEACQR,GRADE,COMMENT)')
USE    BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
      #JSMMSG #SKILL)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
***** Add fields to data source
CHANGE  FIELD(#EMPNO) TO(A1001)
CHANGE  FIELD(#GIVENAME) TO(JOANNE)
CHANGE  FIELD(#SURNAME) TO(JONES)
EXECUTE  SUBROUTINE(FIELDS)
CHANGE  FIELD(#JSMCMD) TO('PUT OBJECT(*FIELD) SERVICE_LIS
      T')
USE    BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET
      #JSMMSG #FLDLST)
***** Send data source to client
USE    BUILTIN(JSM_COMMAND) WITH_ARGS('SEND OBJECT(*DA
      TA') TO_GET(#JSMSTS #JSMMSG)
EXECUTE  SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

```

```

***** 'Close service'
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
***** SUB ROUTINES
SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSMMSG
    D))
IF    COND(#JSMSTS *NE OK')
***** Close service and send the HTTP response
USE    BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
ENDIF
ENDROUTINE
SUBROUTINE NAME(FIELDS)
CHANGE  FIELD(#SNAME) TO(EMPNO)
ADD_ENTRY TO_LIST(#FLDLST)
CHANGE  FIELD(#SNAME) TO(GIVENAME)
ADD_ENTRY TO_LIST(#FLDLST)
CHANGE  FIELD(#SNAME) TO(SURNAME)
ADD_ENTRY TO_LIST(#FLDLST)
ENDROUTINE
SUBROUTINE NAME(BLDMAP)
CHANGE  FIELD(#SNAME) TO(EMPNO)
CHANGE  FIELD(#LNAME) TO(EMPLOYEE_ID)
ADD_ENTRY TO_LIST(#MAPLST)
CHANGE  FIELD(#SNAME) TO(SURNAME)
CHANGE  FIELD(#LNAME) TO(*BLANK)
ADD_ENTRY TO_LIST(#MAPLST)
CHANGE  FIELD(#SNAME) TO(GIVENAME)
CHANGE  FIELD(#LNAME) TO(FIRST_NAME)
ADD_ENTRY TO_LIST(#MAPLST)
CHANGE  FIELD(#SNAME) TO(SKILCODE)
CHANGE  FIELD(#LNAME) TO(SKILL)
ADD_ENTRY TO_LIST(#MAPLST)
CHANGE  FIELD(#SNAME) TO(DATEACQR)
CHANGE  FIELD(#LNAME) TO(ACQUIRED)
ADD_ENTRY TO_LIST(#MAPLST)
CHANGE  FIELD(#SNAME) TO(GRADE)
CHANGE  FIELD(#LNAME) TO(*BLANK)
ADD_ENTRY TO_LIST(#MAPLST)
CHANGE  FIELD(#SNAME) TO(COMMENT)

```

```
CHANGE FIELD(#LNAME) TO(*BLANK)
ADD_ENTRY TO_LIST(#MAPLST)
ENDROUTINE
SUBROUTINE NAME(SKILL)
CHANGE FIELD(#SKILCODE) TO(RPG)
CHANGE FIELD(#DATEACQR) TO(991202)
CHANGE FIELD(#GRADE) TO(A)
CHANGE FIELD(#COMMENT) TO('Good knowledge')
ADD_ENTRY TO_LIST(#SKILL)
CHANGE FIELD(#SKILCODE) TO(CL)
CHANGE FIELD(#DATEACQR) TO(981102)
CHANGE FIELD(#GRADE) TO(B)
CHANGE FIELD(#COMMENT) TO('Improvement required')
ADD_ENTRY TO_LIST(#SKILL)
ENDROUTINE
***** End of RDML commands *****
```

ZipService Example

This is a simple function to create a zip archive file and copy the contents of a nominated directory into the newly created zip file.

* Uses Integrator Services: ZIPSERVICE

* Loads ZIPService service then zips a nominated
* directory into an archive file.

* Beginning of RDML commands *****

```
FUNCTION OPTIONS(*DIRECT)
```

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
```

```
DEFINE FIELD(#ZIPDIR) TYPE(*CHAR) LENGTH(256) LABEL('Zip direc
```

```
DEFINE FIELD(#ZIPFIL) TYPE(*CHAR) LENGTH(256) LABEL('Zip file p
```

* Open service

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMSTS #JSMMSG)
```

* Load service

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVI
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMSTS #JSMMSG)
```

```
MESSAGE MSGTXT('ZIPService loaded')
```

```
BEGIN_LOOP
```

* request name of folder to be zipped and target zip file

```
CHANGE FIELD(#STD_INSTR) TO('Type zip directory and zip file name, p
```

```
REQUEST FIELDS((#STD_INSTR *L003 *P002 *OUTPUT *NOID) (#ZIPI
```

```
IF_KEY WAS(*MENU)
```

```
* Close service
EXECUTE SUBROUTINE(DISCONNECT)
MENU

ENDIF

* create the zip file
EXECUTE SUBROUTINE(MAKEZIP)

END_LOOP

* Zips the nominated directory
SUBROUTINE NAME(MAKEZIP)

* Create the specified zip file
CHANGE FIELD(#JSMCMD) TO(CREATE)
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(FILE #ZIPFIL)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMSTS #JSMMSG)

* Add the contents of the specified folder
CHANGE FIELD(#JSMCMD) TO(ADD)
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(PATH #ZIPDIR)
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(BASE '*CURRENT'
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMSTS #JSMMSG)

* Close the zip file
USE BUILTIN(JSM_COMMAND) WITH_ARGS(CLOSE) TO_GET(#JSMS
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMSTS #JSMMSG)

* Confirm zip is complete
MESSAGE MSGTXT('Directory has been successfully zipped')

ENDROUTINE

SUBROUTINE NAME(DISCONNECT)

* Unload service
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TO_GET
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
* Close service
```

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
ENDROUTINE
```

```
* Build JSM commands
```

```
SUBROUTINE NAME(KEYWRD) PARS((#KEYWORD *RECEIVED) (#
DEFINE FIELD(#KEYWORD) REFFLD(#STD_TEXT)
DEFINE FIELD(#KEYW_VAL1) REFFLD(#STD_TEXTL)
```

```
USE BUILTIN(BCONCAT) WITH_ARGS(#JSMCMD #KEYWORD) TO_GET
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD '(' #KEYW_VAL1 ')')
```

```
ENDROUTINE
```

```
* Check the JSM return status
```

```
SUBROUTINE NAME(CHECK) PARS((#JSMSTS *RECEIVED) (#JSMMSG
```

```
IF COND('#JSMSTS *NE OK')
```

```
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMSTS ':' #JSMMSG) TO_GET
MENU MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#STD_TEXTL)
```

```
ENDIF
```

```
ENDROUTINE
```

PDFSpoolFileService Example

This function allows connection to an IBM i and converts the first 3 spool files from the nominated output queue to PDF documents on the root directory of the active JSM Instance.

- The Server, User and Password are optional if the active JSM Server is located on the same IBM i you want to retrieve spool file information from.
- If the User is NOT specified, the option of *ALL will be used to retrieve spool files. THIS MAY BE A PERFORMANCE OVERHEAD IF YOU NOMINATE AN OUTPUT QUEUE WITH A LARGE NUMBER OF SPOOL FILES.

Modify the default values to be displayed to suit your site.

* Uses Integrator Services: PDFSPOOLFILESERVICE

* This function connects to an IBM i & converts
* the 1st 3 spool files from the nominated outq to PDF
* documents on the root dir of the active JSM Instance.

* Server, User & Password are optional if the active JSM
* Server is located on the same IBM i as spool files

* If the User is NOT specified the option of *ALL is used
* THIS MAY BE A PERFORMANCE OVERHEAD

* Modify the default values to suit your site.

* Beginning of RDML commands *****

FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)

DEFINE FIELD(#JSMSERVER) TYPE(*char) LENGTH(20)
DEFINE FIELD(#JSMUSER) TYPE(*CHAR) LENGTH(010) LABEL('User')

```
DEFINE FIELD(#JSMPSWRD) TYPE(*char) LENGTH(10)
DEFINE FIELD(#JSMLIB) TYPE(*CHAR) LENGTH(010) LABEL('Library'
DEFINE FIELD(#JSMOUTQ) TYPE(*char) LENGTH(10)
```

```
DEFINE FIELD(#FLENAM) TYPE(*CHAR) LENGTH(010)
DEFINE FIELD(#FLENUM) TYPE(*CHAR) LENGTH(006)
DEFINE FIELD(#JOBNAME) TYPE(*CHAR) LENGTH(10)
DEFINE FIELD(#JOBUSER) REFFLD(#JOBNAME)
DEFINE FIELD(#JOBNUMBER) TYPE(*CHAR) LENGTH(006)
DEFINE FIELD(#SPLFILSTS) TYPE(*CHAR) LENGTH(010)
```

```
DEFINE FIELD(#PDFKEY) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#PDFVALUE) TYPE(*CHAR) LENGTH(050)
```

```
DEFINE FIELD(#DOCNME) TYPE(*CHAR) LENGTH(150)
DEFINE FIELD(#DOCCNT) TYPE(*DEC) LENGTH(003) DECIMALS(0)
```

```
DEF_LIST NAME(#PDFLST) FIELDS((#PDFKEY *INP) (#PDFVALUE *IN
```

```
DEF_LIST NAME(#SPOOLST) FIELDS(#FLENAM #FLENUM #JOBNAM
```

* set default values for screen

```
CHANGE FIELD(#STD_INSTR) TO('Set your appropriate information then
CHANGE FIELD(#JMSERVER) TO(ISERIES01)
CHANGE FIELD(#JSMUSER) TO(USER)
CHANGE FIELD(#JSMPSWRD) TO(PASSWORD)
CHANGE FIELD(#JSMLIB) TO(QUSRSYS)
CHANGE FIELD(#JSMOUTQ) TO(QEZJOBLOG)
```

* Connect to JSM Server

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* Load the PDF service

```
CHANGE FIELD(#JSMCMD) TO('SERVICE_LOAD SERVICE(PDFSPPOOL
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
MESSAGE MSGTXT('PDFSpoolFileService loaded')
```

BEGIN_LOOP

* Format and Process JSM commands

REQUEST FIELDS((#STD_INSTR *L004 *P002 *OUTPUT *NOID) (#JSM

IF_KEY WAS(*MENU)

* Close service

EXECUTE SUBROUTINE(DISCONNECT)

MENU

ENDIF

* Connect to IBM i server

EXECUTE SUBROUTINE(CONNECT)

* Get list of spool files to process

EXECUTE SUBROUTINE(LIST)

* Create PDF files

EXECUTE SUBROUTINE(PDF)

END_LOOP

* Connect to host database

SUBROUTINE NAME(CONNECT)

* Connect to the required IBM i host

USE BUILTIN(TCONCAT) WITH_ARGS('CONNECT HOST(' #JMSERVE

USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #JSMPSSWRD ')) TC

USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS

EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

MESSAGE MSGTXT('JSM Server connected')

ENDROUTINE

* Generate a list of spool files from selected location

SUBROUTINE NAME(LIST)

```
MESSAGE MSGTXT('Searching output queue ...') TYPE(*STATUS)
USE BUILTIN(TCONCAT) WITH_ARGS('LIST LIBRARY(' #JSMLIB ') QU
```

```
IF COND('#JSMUSER *NE *BLANK')
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #JSMUSER) TO_GET
ELSE
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD '*ALL') TO_GET(#JS
ENDIF
```

```
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD ') USERDATA(*ALL)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
MESSAGE MSGTXT('List of spool files generated')
```

```
ENDROUTINE
```

```
* Create a PDF file from selected spool file
SUBROUTINE NAME(PDF)
```

```
MESSAGE MSGTXT('PDFs files being generated ...') TYPE(*STATUS)
```

```
* Create an entry for each consistent keyword
CLR_LIST NAMED(#PDFLST)
EXECUTE SUBROUTINE(ADDKEYWRD) WITH_PARMS(JOBNUMBER
EXECUTE SUBROUTINE(ADDKEYWRD) WITH_PARMS(AUTHOR 'LAI
EXECUTE SUBROUTINE(ADDKEYWRD) WITH_PARMS(TITLE 'PDF ex
EXECUTE SUBROUTINE(ADDKEYWRD) WITH_PARMS(SUBJECT 'Gen
EXECUTE SUBROUTINE(ADDKEYWRD) WITH_PARMS(CREATOR *FU
EXECUTE SUBROUTINE(ADDKEYWRD) WITH_PARMS(PAGESIZE '*A
EXECUTE SUBROUTINE(ADDKEYWRD) WITH_PARMS(MARGIN '15')
CHANGE FIELD(#DOCCNT) TO(0)
```

```
SELECTLIST NAMED(#SPOOLLST)
```

```
* build a unique PDF file name
CHANGE FIELD(#DOCCNT) TO('#DOCCNT + 1')
SUBSTRING FIELD(#DOCCNT) INTO_FIELD(#DOCNME)
USE BUILTIN(TCONCAT) WITH_ARGS('PDFTST' #DOCNME '.pdf') TO_(
```

* Build the CREATE command using list details

```
USE BUILTIN(TCONCAT) WITH_ARGS('CREATE SERVICE_LIST(PDFK  
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #FLENUM ') JOBNA  
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD #JOBUSER ') TO_G
```

* Create PDF

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)  
USE BUILTIN(BCONCAT) WITH_ARGS('PDF document' #DOCNME 'creat  
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#STD_TEXTL
```

* Only create PDF files for first 3 spool files

```
LEAVE IF('#DOCCNT = 3')
```

```
ENDSELECT
```

```
ENDROUTINE
```

```
SUBROUTINE NAME(DISCONNECT)
```

* Unload service

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TC  
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

* Close service

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)  
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
ENDROUTINE
```

* Keywords to list

```
SUBROUTINE NAME(ADDKEYWRD) PARMS((#PDFKEY *RECEIVED)
```

```
ADD_ENTRY TO_LIST(#PDFLST)
```

```
ENDROUTINE
```

* Check the JSM return status

SUBROUTINE NAME(CHECK) PARMs((#JSMSTS *RECEIVED) (#JSM

IF COND('#JSMSTS *NE OK')

USE BUILTIN(TCONCAT) WITH_ARGS(#JSMSTS ' : ' #JSMMSG) TO_GE
MENU MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#STD_TEXTL)

ENDIF

ENDROUTINE

* End of RDML commands *****

SVFileService Example

This simple function allows data to be read from or written to a separated variable file.

- * Uses Integrator Services: SVFILESERVICE
- * This function reads CSV file information into a list
- * or write information to a CSV file from a list

* Beginning of RDML commands *****

```
FUNCTION OPTIONS(*DIRECT)
```

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
```

```
DEFINE FIELD(#SVFIL) TYPE(*CHAR) LENGTH(80) LABEL('File path:')
```

```
DEFINE FIELD(#LINENUM) TYPE(*DEC) LENGTH(7) DECIMALS(0) CC
```

```
DEFINE FIELD(#PARTNUM) TYPE(*CHAR) LENGTH(7) COLHDG('Part')
```

```
DEFINE FIELD(#PARTDSC) REFFLD(#STD_DESC)
```

```
DEFINE FIELD(#PARTAMT) TYPE(*DEC) LENGTH(9) DECIMALS(2) CC
```

```
DEFINE FIELD(#PARTQTY) TYPE(*DEC) LENGTH(7) DECIMALS(0) CC
```

```
DEF_LIST NAME(#ORDLIST) FIELDS(#LINENUM #PARTNUM #PARTD
```

```
DEF_LIST NAME(#ORDLISTW) FIELDS(#LINENUM #PARTNUM #PART
```

* Open service

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* Load service

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVI
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
MESSAGE MSGTXT('SVFileService loaded')
```

```
EXECUTE SUBROUTINE(CLEARLST)
```

```
BEGIN_LOOP
```

* request name of folder to be zipped and target zip file
CHANGE FIELD(#STD_INSTR) TO("Type the SV file name to be read or w

REQUEST FIELDS((#STD_INSTR *L003 *P002 *OUTPUT *NOID) (#SVE
(06 'Write' *NEXT *NONE)(07 'Clear'))

IF_KEY WAS(*MENU)

* Close service

EXECUTE SUBROUTINE(DISCONNECT)

MENU

ENDIF

* read the SV file

IF_KEY WAS(*USERKEY1)

EXECUTE SUBROUTINE(READSV)

ENDIF

* write to the SV file

IF_KEY WAS(*USERKEY2)

EXECUTE SUBROUTINE(WRITESV)

ENDIF

* clear the list information

IF_KEY WAS(*USERKEY3)

EXECUTE SUBROUTINE(CLEARLST)

ENDIF

END_LOOP

* Read the nominated SV file

* (columns must match the working list definition)

SUBROUTINE NAME(READSV)

CLR_LIST NAMED(#ORDLISTW)

CLR_LIST NAMED(#ORDLIST)

* Read the SV order file

```
CHANGE FIELD(#JSMCMD) TO('READ SERVICE_LIST(LINENUM,PART
EXECUTE SUBROUTINE(KEYWRD) WITH_PARM(SFILE #SVFIL)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARM(S#JSMSTS #JSMMSG)
```

```
SELECTLIST NAMED(#ORDLISTW)
```

```
ADD_ENTRY TO_LIST(#ORDLIST) WITH_MODE(*ADD)
```

```
ENDSELECT
```

```
* add a blank line for new details
```

```
CHANGE FIELD(#ORDLIST) TO(*DEFAULT)
```

```
CHANGE FIELD(#LINENUM) TO(#LISTCOUNT)
```

```
BEGIN_LOOP TO(5)
```

```
CHANGE FIELD(#LINENUM) TO('#LINENUM + 1')
```

```
ADD_ENTRY TO_LIST(#ORDLIST) WITH_MODE(*ADD)
```

```
END_LOOP
```

```
* Confirm file read is complete
```

```
MESSAGE MSGTXT('File has been successfully read')
```

```
ENDROUTINE
```

```
* Write to the nominated SV file
```

```
* (columns must match the working list definition)
```

```
SUBROUTINE NAME(WRITESV)
```

```
CLR_LIST NAMED(#ORDLISTW)
```

```
SELECTLIST NAMED(#ORDLIST)
```

```
IF COND('#PARTNUM *NE *BLANK')
```

```
ADD_ENTRY TO_LIST(#ORDLISTW)
```

```
ENDIF
```

ENDSELECT

* Create the specified zip file

```
CHANGE FIELD(#JSMCMD) TO('WRITE SERVICE_LIST(LINENUM,PAR
EXECUTE SUBROUTINE(KEYWRD) WITH_PARM(SFILE #SVFIL)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARM(S#JSMSTS #JSMMSG)
```

* Confirm write of file is complete

```
MESSAGE MSGTXT('Order has been successfully written to file')
```

ENDROUTINE

* Clear list

```
SUBROUTINE NAME(CLEARLST)
```

```
CLR_LIST NAMED(#ORDLIST)
```

```
BEGIN_LOOP USING(#LINENUM) TO(10)
ADD_ENTRY TO_LIST(#ORDLIST) WITH_MODE(*ADD)
END_LOOP
```

ENDROUTINE

```
SUBROUTINE NAME(DISCONNECT)
```

* Unload service

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TC
```

* Close service

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARM(S#JSMSTS #JSMMSG)
```

ENDROUTINE

* Build JSM commands

```
SUBROUTINE NAME(KEYWRD) PARM(S((#KEYWORD *RECEIVED) (#
```

```
DEFINE FIELD(#KEYWORD) REFFLD(#STD_TEXT)
DEFINE FIELD(#KEYW_VAL1) REFFLD(#STD_TEXTL)
```

```
USE BUILTIN(BCONCAT) WITH_ARGS(#JSMCMD #KEYWORD) TO_GI
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD '(' #KEYW_VAL1 ')')
```

```
ENDROUTINE
```

```
* Check the JSM return status
```

```
SUBROUTINE NAME(CHECK) PARS((#JSMSTS *RECEIVED) (#JSMM
```

```
IF COND('#JSMSTS *NE OK')
```

```
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMSTS ' : ' #JSMMSG) TO_GE
```

```
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#STD_TEXTL
```

```
ENDIF
```

```
ENDROUTINE
```

ExcelReadService Example

* Uses Integrator Services: EXCELREADSERVICE

* This function reads an EXCEL file into a list

* or writes information to an EXCEL file from a list

* Beginning of RDML commands *****

```
FUNCTION OPTIONS(*DIRECT)
```

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
```

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
```

```
DEFINE FIELD(#EXCELFIL) TYPE(*CHAR) LENGTH(79) LABEL('File p
```

```
DEFINE FIELD(#EXCELSHT) TYPE(*CHAR) LENGTH(25) LABEL('Work
```

```
DEFINE FIELD(#LINENUM) TYPE(*DEC) LENGTH(7) DECIMALS(0) CC
```

```
DEFINE FIELD(#PARTNUM) TYPE(*CHAR) LENGTH(7) COLHDG('Part')
```

```
DEFINE FIELD(#PARTDSC) REFFLD(#STD_DESC)
```

```
DEFINE FIELD(#PARTAMT) TYPE(*DEC) LENGTH(9) DECIMALS(2) CC
```

```
DEFINE FIELD(#PARTQTY) TYPE(*DEC) LENGTH(7) DECIMALS(0) CC
```

```
DEF_LIST NAME(#ORDLIST) FIELDS(#LINENUM #PARTNUM #PARTD
```

```
DEF_LIST NAME(#ORDLISTW) FIELDS(#LINENUM #PARTNUM #PART
```

* Open service

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* Load service

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVI
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
MESSAGE MSGTXT('ExcelReadService loaded')
```

```
EXECUTE SUBROUTINE(CLEARLST)
```

```
BEGIN_LOOP
```

* request name of folder to be zipped and target zip file
CHANGE FIELD(#STD_INSTR) TO("Type the EXCEL file name to be read

REQUEST FIELDS((#STD_INSTR *L003 *P002 *OUTPUT *NOID) (#EXC
(06 'Write' *NEXT *NONE)(07 'Clear'))

IF_KEY WAS(*MENU)

* Close service

EXECUTE SUBROUTINE(DISCONNECT)
MENU
ENDIF

* read the EXCEL file

IF_KEY WAS(*USERKEY1)
EXECUTE SUBROUTINE(READEXCEL)
ENDIF

* write to the EXCEL file

IF_KEY WAS(*USERKEY2)
EXECUTE SUBROUTINE(WRITEEXCEL)
ENDIF

* clear the list information

IF_KEY WAS(*USERKEY3)
EXECUTE SUBROUTINE(CLEARLST)
ENDIF

END_LOOP

* Read the nominated EXCEL file

* (columns must match the working list definition)

SUBROUTINE NAME(READEXCEL)

CLR_LIST NAMED(#ORDLISTW)

CLR_LIST NAMED(#ORDLIST)

* Open the EXCEL order file

CHANGE FIELD(#JSMCMD) TO(OPEN)

```
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(FILE #EXCELFIL)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

* Read the EXCEL order file

```
CHANGE FIELD(#JSMCMD) TO('READ SERVICE_LIST(LINENUM,PAR
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(R1C1 '2,1')
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(R2C2 '3,0')
IF COND('#EXCELSHT *NE *BLANK')
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMS(SHEET #EXCELSH
ENDIF
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
SELECTLIST NAMED(#ORDLISTW)
ADD_ENTRY TO_LIST(#ORDLIST) WITH_MODE(*ADD)
ENDSELECT
```

* add a blank line for new details

```
CHANGE FIELD(#ORDLIST) TO(*DEFAULT)
CHANGE FIELD(#LINENUM) TO(#LISTCOUNT)
```

```
BEGIN_LOOP TO(5)
CHANGE FIELD(#LINENUM) TO('#LINENUM + 1')
ADD_ENTRY TO_LIST(#ORDLIST) WITH_MODE(*ADD)
END_LOOP
```

```
EXECUTE SUBROUTINE(CLOSE)
```

* Confirm file read is complete

```
MESSAGE MSGTXT('File has been successfully read')
```

```
ENDROUTINE
```

* Write to the nominated EXCEL file

* (columns must match the working list definition)

```
SUBROUTINE NAME(WRITEEXCEL)
```

```
CLR_LIST NAMED(#ORDLISTW)
```

```
SELECTLIST NAMED(#ORDLIST)
```

```
IF COND('#PARTNUM *NE *BLANK')  
ADD_ENTRY TO_LIST(#ORDLISTW)  
ENDIF
```

```
ENDSELECT
```

```
* Open the EXCEL order file
```

```
CHANGE FIELD(#JSMCMD) TO('OPEN MODE(*WRITE)')  
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(FILE #EXCELFIL)  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS  
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMSTS #JSMMSG)
```

```
* Create the specified EXCEL file
```

```
CHANGE FIELD(#JSMCMD) TO('WRITE SERVICE_LIST(LINENUM,PAR  
IF COND('#EXCELSHT *NE *BLANK')  
EXECUTE SUBROUTINE(KEYWRD) WITH_PARMs(SHEET #EXCELSH  
ENDIF  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS  
EXECUTE SUBROUTINE(CHECK) WITH_PARMs(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CLOSE)
```

```
* Confirm write of file is complete
```

```
MESSAGE MSGTXT('Order has been successfully written to file')
```

```
ENDROUTINE
```

```
* Close all open worksheets
```

```
SUBROUTINE NAME(CLOSE)  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(CLOSE) TO_GET(#JSMST  
ENDROUTINE
```

```
* Clear list
```

```
SUBROUTINE NAME(CLEARLST)  
CLR_LIST NAMED(#ORDLIST)
```

```
BEGIN_LOOP USING(#LINENUM) TO(10)
ADD_ENTRY TO_LIST(#ORDLIST) WITH_MODE(*ADD)
END_LOOP
```

```
ENDROUTINE
```

```
SUBROUTINE NAME(DISCONNECT)
```

```
* Unload service
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_UNLOAD') TC
```

```
* Close service
```

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
ENDROUTINE
```

```
* Build JSM commands
```

```
SUBROUTINE NAME(KEYWRD) PARS((#KEYWORD *RECEIVED) (#
```

```
DEFINE FIELD(#KEYWORD) REFFLD(#STD_TEXT)
```

```
DEFINE FIELD(#KEYW_VAL1) REFFLD(#STD_TEXTL)
```

```
USE BUILTIN(BCONCAT) WITH_ARGS(#JSMCMD #KEYWORD) TO_GI
```

```
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMCMD '(' #KEYW_VAL1 ')')
```

```
ENDROUTINE
```

```
* Check the JSM return status
```

```
SUBROUTINE NAME(CHECK) PARS((#JSMSTS *RECEIVED) (#JSM
```

```
IF COND(#JSMSTS *NE OK')
```

```
USE BUILTIN(TCONCAT) WITH_ARGS(#JSMSTS ' : ' #JSMMSG) TO_GE
```

```
MESSAGE MSGID(DCM9899) MSGF(DC@M01) MSGDTA(#STD_TEXTL
```

```
ENDIF
```

```
ENDROUTINE
```

SQLService Examples

To help clarify the use of the SQLService, particularly the different options in the SET PARAMETER (value), the EXECUTE and the READ commands, three examples than normal have been supplied.

These are:

- [Example 1 - Use of the SET PARAMETER\(*SQL\) command](#)
- [Example 2 - Use of the SET PARAMETER\(*LIST\)](#)
- [Example 3 - READ command receiving the result set from query](#)

Example 1 - Use of the SET PARAMETER(*SQL) command

This example demonstrates how to use the SQLService to query a database. It will use the SET PARAMETER(*SQL) command to create an SQL statement in advance instead of writing it into the EXECUTE command. A note is included to show how you could achieve the same query without using the SET PARAMETER(*SQL) and simply writing the SQL statement into the EXECUTE command.

Note that this example connects to an IBM i database called JSMJDBC (since this is an IBM i example, the database name is a library name). The driver name used in the CONNECT command corresponds to the driver name and path defined in the SQLService properties file. The file being accessed is called TBLNAME and it consists of the fields ID, NAME, AGE, SALARY, DEPT, and GRADE.

The steps that you will follow are:

1. Define the fields to handle the messages to be returned from the JSMCOMMANDs.
2. Define some fields that will hold the values to be returned from the query.
3. Define a working list whose columns are the fields created in step 2. This working list will eventually hold the result set from the query.
4. Define a field that will hold the SQL statement.
5. Define a working list that will hold the SQL statement. This will be a single column list and the field used will be that defined in Step 4.
6. Start JSM, LOAD the SQLService, then CONNECT to the database driver.
7. Prepare the SQL statement, then place it into a working list.
8. Use the SET command to store the SQL parameter. You will note that the keyword PARAMETER(*SQL) was used to indicate that this command is to provide a working list that will hold the SQL statement for the EXECUTE command to be executed later. Also note that there is a SERVICE_LOAD keyword associated with this command to specify the field in the working list that holds the SQL statement. The field name specified here must be the same as that defined in the working list in the TO_GET portion of this JSM command.
9. The next step is to actually EXECUTE the command. You will note in this example, since an already prepared SQL statement is being used, a value of

*SQLPARAMETER is specified for the QUERY keyword. You will also notice that a service list is used as a part of this command - this is used to receive the values back into. The columns defined in the service list here must match the columns defined in the working list for the JSM_COMMAND Built In Function.

10. After displaying the results, you will disconnect the service and then close JSM.

- * This example demonstrates how to use the SQLService to
- * query a database. The SET PARAMETER(*SQL)
- * command is used to create an SQL statement
- * in advance instead of writing it into the EXECUTE command.
- * A note is included to show how you could achieve the
- * same query simply by writing the SQL statement
- * into the EXECUTE command without using the SET PARAMETER(*SQL).
- *

- * Note 1: This example connects to a
- * IBM i database called JSMJDBC. (Since this is a
- * IBM i example, the database name is a library name.)
- * The driver name used in the CONNECT command corresponds
- * to the driver name and path defined in the SQLService
- * properties file. The file being accessed is called
- * TBLNAME and it consists of the fields ID, NAME, AGE,
- * SALARY, DEPT, and GRADE.

FUNCTION OPTIONS(*DIRECT)

- * 1. Define the fields to handle the messages to be
- * returned from the JSMCOMMANDs

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
```

- * 2. Define some fields that will hold the values to be
- * returned from the query

```
DEFINE FIELD(#COL1) TYPE(*CHAR) LENGTH(010)
DEFINE FIELD(#COL2) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#COL3) TYPE(*DEC) LENGTH(008) DECIMALS(0)
DEFINE FIELD(#COL4) TYPE(*DEC) LENGTH(012) DECIMALS(2)
```

- * 3. Define a working list whose columns are the fields
- * created in step 2. This working list will eventually
- * hold the result set from the query

```
DEF_LIST NAME(#WRKLST) FIELDS(#COL1 #COL2 #COL3 #COL4) TY
DEF_LIST NAME(#BRWLST) FIELDS(#COL1 #COL2 #COL3 #COL4)
```

- * 4. Define a field that will hold the SQL statement

```
DEFINE FIELD(#COLCMD) TYPE(*CHAR) LENGTH(100)
```

- * 5. Define a working list that will hold the SQL
- * statement. This will be a single column list and the
- * field used will be that defined in Step 4

```
DEF_LIST NAME(#WRKCMD) FIELDS(#COLCMD) TYPE(*WORKING)
```

- * 6. Start OPEN JSM, LOAD the SQLService, then
- * CONNECT to the database driver

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVI
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
CHANGE FIELD(#JSMCMD) TO('CONNECT DRIVER(DB2) DATABASE(
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

- * 7. Prepare the SQL statement then place it in
- * a working list

```
CHANGE FIELD(#COLCMD) TO('SELECT ID,NAME,AGE,SALARY FRO
ADD_ENTRY TO_LIST(#WRKCMD)
```

- * 8. Use the SET command to store the SQL parameter.
- * You will note that the PARAMETER(*SQL) keyword
- * is used to indicate that this command provides
- * a working list that will hold the SQL statement
- * for the EXECUTE command to be executed later

```
CHANGE FIELD(#JSMCMD) TO('SET PARAMETER(*SQL) SERVICE_LI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMSTS #JSMMSG)
```

- * 9. The next step is to actually EXECUTE the comand.
- * You will note in this example, since an already
- * prepared SQL statement is being used, a value of
- * *SQLPARAMETER is specified for the QUERY keyword.
- * You will also notice that a service list is used as a
- * part of this command - this is used to receive the
- * values back into. The columns defined in the service
- * list here must match the columns defined in the working
- * list for the JSM_COMMAND Built In Function.

```
CHANGE FIELD(#JSMCMD) TO('EXECUTE QUERY(*SQLPARAMETER  
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS  
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMSTS #JSMMSG)
```

* NOTE:

- * Another way to express this SQL statement could have
- * been to place it directly into the EXECUTE command.
- * With this approach, you would remove the SET command
- * earlier in the program and rewrite the EXECUTE command
- * as follows:
- * CHANGE FIELD(#JSMCMD) TO('EXECUTE QUERY(SELECT ID,NAM
* AGE,SALARY FROM TBLNAME) SERVICE_LIST(COL1,COL2,COL3
* ,COL4)')

```
SELECTLIST NAMED(#WRKLST)  
ADD_ENTRY TO_LIST(#BRWLST)  
ENDSELECT  
DISPLAY BROWSELIST(#BRWLST)
```

- * 10. After displaying the results, you will disconnect
 - * the service and then close JSM.
- ```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(DISCONNECT) TO_GET(
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMSTS #JSMMSG)
```

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMSTS #JSMMSG)
```

\* SUB ROUTINES

```
SUBROUTINE NAME(CHECK) PARS((#JSMSTS *RECEIVED) (#JSM
```

```
IF COND('#JSMSTS *NE OK')
```

```
DISPLAY FIELDS(#JSMMSG)
```

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
MENU MSGTXT('Java service error has occurred')
```

```
ENDIF
```

```
ENDROUTINE
```

## **Example 2 - Use of the SET PARAMETER(\*LIST)**

The purpose of this example is to demonstrate how to use the SET PARAMETER(\*LIST) command to prepare values to be inserted into a table. The example is extended to show the syntax required for an INSERT and an UPDATE. The program is rather futile in that it inserts a record then immediately updates the same record, but the real purpose of this is to demonstrate how the commands should look. This example also demonstrates how the SET command can be used a number of times to achieve different objectives. In this case there are four SET commands: one to set the commitment control settings, one to prepare an SQL command in advance using the SET PARAMETER(\*SQL) command, and of course the ones used to prepare the list of values to be inserted and updated.

Note that in this example connection is to an IBM i database called JSMJDBC (since this is an IBM i example, the database name is a library name). The driver name used in the CONNECT command corresponds to the driver name and path defined in the SQLService properties file. The file being accessed is called TBLNAME and it consists of the fields ID, NAME, AGE, SALARY, DEPT, and GRADE.

The steps are:

1. Define the fields to handle the messages to be returned from the JSMCOMMANDs.
2. Define a field that will hold the SQL statement.
3. Define a working list that will hold the SQL statement. This will be a single column list and the field used will be that defined in Step 4.
4. The values that are to be inserted into the remote table will first be placed into a working list. The working list will have one column for each field that needs to be inserted into the file. So first, the fields for these columns need to be defined - they must match the type and length of the fields in the table. Once defined, the working list can be defined. In this example we will be inserting the ID, NAME, AGE, and SALARY.
5. Another working list is defined to hold the fields for the update. In this case, the SALARYs are to be updated, so this working list will contain the SALARY in one column and the ID (the key) in another. The order in which these columns are placed is important and is explained later.
6. Start JSM, LOAD the SQLService, then CONNECT to the database driver.

7. Issue a SET command to configure the commitment control requirements.
8. We will next prepare the SQL statement then place it into a working list.
9. The SET command will be used to store the SQL parameter. You will note that:
  - the keyword PARAMETER(\*SQL) will be used to indicate that this command is being used to provide a working list that will hold the SQL statement for the EXECUTE command to be executed later.
  - there is a SERVICE\_LOAD keyword associated with this command. This specifies the field in the working list that holds the SQL statement. The field name specified here must be the same as that defined in the working list in the TO\_GET portion of this JSM command.
  - the values keyword of the INSERT has four questions marks (?) in it. These '?' represent the columns in the WRKLSTINS working list, and they are matched up to the '?' in the order they appear in the working list. So in this example, ID values go to the first '?', NAME values go to the second '?' and so forth;
10. Now you will specify some new records to be entered into the database. Notice that each new record appears on a new line of the list.
11. Once the values are prepared in the list, you need to use the SET PARAMETER(\*LIST) command so that the EXECUTE command is able to make use of this information later. Notice that a SERVICE\_LIST is specified as part of this command. It contains the names of the columns that appear in the WRKLSTINS working list. Also note that this working list is specified in the TO\_GET portion of this JSM command.
12. The final step in this part of the program is to run the EXECUTE command that will insert the records into the table. Since the values have been prepared already, in this command you use the keyword PREPARED(\*SQLPARAMETER).

**Note:** If the SQL statement had not been prepared earlier, then you would have typed it in place of the \*SQLPARAMETER - for example, PREPARED('INSERT INTO .....')
13. As was done for preparing the INSERT data, the updated data is loaded into the WRKLSTUPD working list and then prepared using the SET PARAMETER(\*LIST) command.
14. The EXECUTE command here demonstrates the importance of the column

order when using the PREPARED approach. You will notice in the UPDATE that the first '?' is for the name, and the second '?' is for the ID. Therefore the columns must appear in the WRKLSTUPD working list in the same order as they are to be placed in the SQL statement.

15. COMMIT the results

16. Finally disconnect the service then close JSM.

- \* The purpose of this example is to demonstrate how to
- \* use the SET PARAMETER(\*LIST) command to prepare values
- \* to be inserted into a table. The example is extended to
- \* show the syntax required for an INSERT and an UPDATE.
- \* This program is rather futile in that it inserts a
- \* record then immediately updates the same record, but
- \* the real purpose of this is to demonstrate how the
- \* commands should look. This example also demonstrates
- \* how the SET command can be used a number of times to
- \* achieve different objectives. In this case there are 4
- \* SET commands - one to set the commitment control
- \* settings, one to prepare an SQL command in advance
- \* using the SET PARAMETER(\*SQL) command, and of course
- \* the ones used to prepare the list of values to be
- \* inserted and updated.

- \* Note that in this example connection is to a
- \* IBM i database called JSMJDBC (since this is a
- \* IBM i example, the database name is a library name).
- \* The driver name used in the CONNECT command corresponds
- \* to the driver name and path defined in the SQLService
- \* properties file. The file being accessed is called
- \* TBLNAME and it consists of the fields ID, NAME, AGE,
- \* SALARY, DEPT, and GRADE.

- \* The steps to be followed are embedded in the code.

FUNCTION OPTIONS(\*DIRECT)

- \* 1. Define the fields to handle the messages to be
- \* returned from the JSMCOMMANDS

DEFINE FIELD(#JSMSTS) TYPE(\*CHAR) LENGTH(020)

```
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
```

\* 2. Define a field that will hold the SQL statement

```
DEFINE FIELD(#COLCMD) TYPE(*CHAR) LENGTH(100)
```

\* 3. Define a working list that will hold the SQL

\* statement. This will be a single column list and the

\* field used will be that defined in Step 2

```
DEF_LIST NAME(#WRKCMD) FIELDS(#COLCMD) TYPE(*WORKING)
```

\* 4. The values that are to be inserted into the remote

\* table will first be placed into a working list. The

\* working list will have one column for each field we

\* need to insert into the file. So first, the fields need

\* to be defined for these columns- they must match the

\* type and length of the fields in the table. Once

\* defined, the working list can be defined. In this

\* example the ID, NAME, AGE, and SALARY will be inserted.

```
DEFINE FIELD(#COL1) TYPE(*CHAR) LENGTH(010)
```

```
DEFINE FIELD(#COL2) TYPE(*CHAR) LENGTH(020)
```

```
DEFINE FIELD(#COL3) TYPE(*DEC) LENGTH(008) DECIMALS(0)
```

```
DEFINE FIELD(#COL4) TYPE(*DEC) LENGTH(012) DECIMALS(2)
```

```
DEF_LIST NAME(#WRKLISTINS) FIELDS(#COL1 #COL2 #COL3 #COL4)
```

\* 5. Another working list is defined to hold the fields

\* for the update. In this case we want to update the

\* SALARYs, so this working list will contain the SALARY

\* in one column and the ID (the key) in another. The

\* order in which these columns are placed is important

\* and is explained later

```
DEF_LIST NAME(#WRKLISTUPD) FIELDS(#COL3 #COL1) TYPE(*WOR
```

\* 6. Start JSM, LOAD the SQLService, then CONNECT to the

\* database driver

\* 'Open service'

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
```

```
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

\* 'Load service'

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVICE_LOAD')
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
CHANGE FIELD(#JSMCMD) TO('CONNECT DRIVER(DB2) DATABASE(#JSMDB)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMDB)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

\* 7. Issue a SET command to configure the commitment

\* control requirements

```
CHANGE FIELD(#JSMCMD) TO('SET ISOLATION(*READCOMMITTED)
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JSMDB)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

\* 8. You will next prepare the SQL statement then place it

\* into a working list

```
CHANGE FIELD(#COLCMD) TO('INSERT INTO TBLNAME(ID,NAME,ADDRESS)
ADD_ENTRY TO_LIST(#WRKCMD)
```

\* 9. The SET command will be used to store the SQL

\* parameter. You will note that:

\* the keyword PARAMETER(\*SQL) will be used to indicate

\* that this command is to provide a working list that

\* will hold the SQL statement for the EXECUTE command

\* to execute later.

\* - there is a SERVICE\_LOAD keyword associated with

\* this command. This specifies the field in the working

\* list that holds the SQL statement. The \* field name

\* specified here must be the same as that defined in

\* the working list in the TO\_GET portion of this

\* JSM command.

\* - the values keyword of the INSERT has four question

\* marks ('?') in it. These '?' represent the columns in the

\* WRKLIST working list, and they are matched up to the

\* '?' in the order they appear in the working list. So in

\* this example, ID values go to the first '?', NAME

\* values go to the second '?' and so forth

```
CHANGE FIELD(#JSMCMD) TO('SET PARAMETER(*SQL) SERVICE_LOAD')
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

- \* 10. Now specify some new records to be entered into
- \* the database. Notice that each new record appears
- \* on a new line of the list.

```
CHANGE FIELD(#COL1) TO(S2221)
CHANGE FIELD(#COL2) TO(TOM)
CHANGE FIELD(#COL3) TO(45)
CHANGE FIELD(#COL4) TO(35000.60)
ADD_ENTRY TO_LIST(#WRKLSTINS)
```

```
CHANGE FIELD(#COL1) TO(S2222)
CHANGE FIELD(#COL2) TO(SQUIRE)
CHANGE FIELD(#COL3) TO(22)
CHANGE FIELD(#COL4) TO(27000.60)
ADD_ENTRY TO_LIST(#WRKLSTINS)
```

- \* 11. Once the values are prepared in the list, you need
- \* to use the SET PARAMETER(\*LIST) command so that the
- \* EXECUTE command is able to make use of this information
- \* later. Notice that a SERVICE\_LIST is specified as a
- \* part of this command. It contains the names of the
- \* columns that appear in the WRKLSTINS working list. Also
- \* note that this working list is specified in the TO\_GET
- \* portion of this JSM command

```
CHANGE FIELD(#JSMCMD) TO('SET PARAMETER(*LIST) SERVICE_LI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

- \* 12. The final step in this part of the program is to
- \* run the EXECUTE command which will insert the records
- \* into the table. Since the values have been prepared
- \* already, in this command, the keyword
- \* PREPARED(\*SQLPARAMETER) is used.
- \* Note: If the SQL statement had not been prepared earlier
- \* then you would have typed it in place of the
- \* \*SQLPARAMETER - for example,
- \* PREPARED('INSERT INTO .....')

```
CHANGE FIELD(#JSMCMD) TO('EXECUTE PREPARED(*SQLPARAMET
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
* After the EXECUTE the insert is COMMITed
USE BUILTIN(JSM_COMMAND) WITH_ARGS(COMMIT) TO_GET(#JSM
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

\*

13. The next section of code demonstrates how an update  
\* might work. As was done for preparing the INSERT data,  
\* the updated data is loaded into the WRKLSTUPD working  
\* list and the prepared using the SET PARAMETER(\*LIST)  
\* command.

```
CHANGE FIELD(#COL3) TO(123456.99)
CHANGE FIELD(#COL1) TO(S2221)
ADD_ENTRY TO_LIST(#WRKLSTUPD)
```

```
CHANGE FIELD(#COL3) TO(654321.11)
CHANGE FIELD(#COL1) TO(S2222)
ADD_ENTRY TO_LIST(#WRKLSTUPD)
```

```
CHANGE FIELD(#JSMCMD) TO('SET PARAMETER(*LIST) SERVICE_LI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

\* 14. The EXECUTE command here demonstrates the  
\* importance of the column order when using the PREPARED  
\* approach. You will notice in the UPDATE that the  
\* first '?' is for the name, and the second '?' is for  
\* the ID. Therefore the columns must appear in the  
\* WRKLSTUPD working list in the same order as they are to  
\* be placed in the SQL statement.

```
CHANGE FIELD(#JSMCMD) TO('EXECUTE PREPARED(UPDATE TBLN.
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

\* 15. COMMIT the results

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(COMMIT) TO_GET(#JSM
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMSTS #JSMMSG)
```

\* 16. Finally disconnect the service then close JSM.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(DISCONNECT) TO_GET
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMSTS #JSMMSG)
```

\* 'Close service'

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARM(#JSMSTS #JSMMSG)
```

\* SUB ROUTINES

```
SUBROUTINE NAME(CHECK) PARM(#JSMSTS *RECEIVED) (#JSM
```

```
IF COND('#JSMSTS *NE OK')
```

```
DISPLAY FIELDS(#JSMMSG)
```

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```
MENU MSGTXT('Java service error has occurred')
```

```
ENDIF
```

```
ENDROUTINE
```

### **Example 3 - READ command receiving the result set from query**

In this SQLService example, you will see how to use the READ command to receive the result set from the query. In using the READ command you must use the SET PARAMETER(\*MAP) command to set up the field column mappings.

Note that in this example connection is to an IBM i database called JSMJDBC (since this is an IBM i example, the database name is a library name).

The driver name used in the CONNECT command corresponds to the driver name and path defined in the SQLService properties file. The file being accessed is called TBLNAME and it consists of the fields ID, NAME, AGE, SALARY, DEPT, and GRADE.

The steps that this example follows are:

1. Define the fields to handle the messages to be returned from the JSMCOMMANDs;
2. Define the fields that will be used to map the table fields to the columns in the results list. The fields defined here should match the length and type of the fields defined in the table being queried.
3. Define the two fields required for the list that is used to hold the field column mappings.
4. Define the list that will hold the column mappings. This should be a two-column list - the first column in this list will hold the column number and the second field will hold the column name.
5. Define two working lists using the fields defined in step 2. These lists will be populated by the READ command from the result list that the query generates. In this example, two columns are used to demonstrate how you can keep going back to the result list to access different columns as and when required. Two browselists are defined with the same shape as these working lists, and these will be used to display the results on a screen. See point 9 for details on the significance of the ENTRYYS value for the working lists.
6. Next start JSM, LOAD the SQLService, then CONNECT to the database driver.
7. Define the column field mappings. This is done by setting #FIELD to the appropriate column and the #COLUMN to the appropriate field name (from the table). Each mapping will be added as a new entry to the #MAPLST working list, and then a SET PARAMETER(\*MAP) command will be issued.

Notice that a SERVICE\_LIST is passed as a part of this command - the fields defined here are those used in the mapping list. The mapping list must also be specified in the TO\_GET portion of the JSM command.

8. EXECUTE the command. In this example, a prepared statement is not used, but is typed directly into the EXECUTE command. Notice how a working list is not supplied with this JSM command. For an EXECUTE QUERY the working list would normally be provided to hold the result list, but in this case the READ command has been chosen instead to access the result list.
9. The next step is to issue the READ command to access the data in the result set. You will notice that this is done twice in this example, each time accessing different columns. The capability demonstrated with this example will be particularly useful in situations where your result list is larger than what a working list can hold, so by reducing the number of columns in the READ you can access the data in more manageable chunks. You will note that the first READ has a keyword SCROLL(\*NO). This allows the second READ to select the columns from the same records selected by the first READ. The second READ has a keyword SCROLL(\*YES), which means that when the next DOWHILE loop begins, the first READ will begin with the next set of records. The number of records read with each READ depends on the size of the working lists, which in this example has been set to 100 - NB ENTRIES(100). Notice that the READ commands include a SERVICE\_LIST - the fields specified with this keyword must be the same as the fields defined in the working list specified in the TO\_GET section of the JSM command.
10. After displaying the results, disconnect the service then close JSM.
  - \* In this SQLService example, you will see how to
  - \* use the READ command to receive the result set from the
  - \* query. In using the READ command you must use the SET
  - \* PARAMETER(\*MAP) command to set up the field column
  - \* mappings.
  
  - \* Note that in this example, connection is to a
  - \* IBM i database called JSMJDBC (since this is a
  - \* IBM i example, the database name is a library name).
  - \* The driver name used in the CONNECT command corresponds
  - \* to the driver name and path defined in the SQLService
  - \* properties file. The file being accessed is called

- \* TBLNAME and it consists of the fields ID, NAME, AGE,
- \* SALARY, DEPT, and GRADE.

\* The steps to follow are embedded in the code:

FUNCTION OPTIONS(\*DIRECT)

- \* 1. Define the fields to handle the messages to be
- \* returned from the JSMCOMMANDS

```
DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)
DEFINE FIELD(#JSMCMD) TYPE(*CHAR) LENGTH(255)
```

- \* 2. Define the fields that will be used to map the table
- \* fields to the columns in the results list. The fields
- \* defined here should match the length and type of the
- \* fields defined in the table being queried

```
DEFINE FIELD(#COL1) TYPE(*CHAR) LENGTH(010)
DEFINE FIELD(#COL2) TYPE(*CHAR) LENGTH(020)
DEFINE FIELD(#COL3) TYPE(*DEC) LENGTH(008) DECIMALS(0)
DEFINE FIELD(#COL4) TYPE(*DEC) LENGTH(012) DECIMALS(2)
```

- \* 3. Define the 2 fields required for the list that is
- \* used to hold the field column mappings.

```
DEFINE FIELD(#FIELD) TYPE(*CHAR) LENGTH(010)
DEFINE FIELD(#COLUMN) TYPE(*CHAR) LENGTH(030)
```

- \* 4. Define the list that will hold the column mappings.
- \* This should be a two column list - the first column in
- \* this list will hold the column name and the second
- \* field will hold the column name

```
DEF_LIST NAME(#MAPLST) FIELDS(#FIELD #COLUMN) TYPE(*WOR
```

- \* 5. Define 2 working lists using the fields defined in
- \* the step 2. These lists will be populated by the READ
- \* command from the result list that the query generates.
- \* In this example, 2 columns are used to demonstrate how
- \* you can keep going back to the result list to access
- \* different columns as and when required. 2 browselists

- \* with the same shape as these working lists are defined,
- \* and these will be used to display the results on a
- \* screen. See point 9 for details on the significance of
- \* the ENTRYS value for the working lists.

```
DEF_LIST NAME(#WRKLST1) FIELDS(#COL1 #COL3) TYPE(*WORKIN
DEF_LIST NAME(#WRKLST2) FIELDS(#COL1 #COL2 #COL4) TYPE(*W
DEF_LIST NAME(#BRWLST1) FIELDS(#COL1 #COL3) ENTRYS(100)
DEF_LIST NAME(#BRWLST2) FIELDS(#COL1 #COL2 #COL4) ENTRYS(
```

- \* 6. Next start JSM, LOAD the SQLService, then CONNECT to
- \* the database driver

```
USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SERVI
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
CHANGE FIELD(#JSMCMD) TO('CONNECT DRIVER(DB2) DATABASE(
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

- \* 7. Define the column field mappings. This is done by
- \* setting #FIELD to the appropriate column and the
- \* #COLUMN to the appropriate field name (from the table).
- \* Each mapping will be added as a new entry to the
- \* #MAPLST working list, and then a SET PARAMETER(\*MAP)
- \* command will be issued. Notice that a SERVICE\_LIST is
- \* passed as a part of this command - the fields defined
- \* here are those used in the mapping list. The mapping
- \* list must also be specified in the TO\_GET portion of
- \* the JSM command.

```
CHANGE FIELD(#FIELD) TO(COL1)
CHANGE FIELD(#COLUMN) TO(ID)
ADD_ENTRY TO_LIST(#MAPLST)
CHANGE FIELD(#FIELD) TO(COL2)
CHANGE FIELD(#COLUMN) TO(NAME)
ADD_ENTRY TO_LIST(#MAPLST)
```

```
CHANGE FIELD(#FIELD) TO(COL3)
CHANGE FIELD(#COLUMN) TO(AGE)
ADD_ENTRY TO_LIST(#MAPLST)
CHANGE FIELD(#FIELD) TO(COL4)
CHANGE FIELD(#COLUMN) TO(SALARY)
ADD_ENTRY TO_LIST(#MAPLST)
```

```
CHANGE FIELD(#JSMCMD) TO('SET PARAMETER(*MAP) SERVICE_LI
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

- \* 8. EXECUTE the command. In this example, a prepared
- \* statement is not used, but is typed directly into the
- \* EXECUTE command. Notice how a working list is not supplied
- \* with this JSM command. For an EXECUTE QUERY the working
- \* list would normally be provided to hold the result list,
- \* but in this case, the READ command has been chosen
- \* to access the result list instead.

```
CHANGE FIELD(#JSMCMD) TO('EXECUTE QUERY(SELECT ID,NAME,
USE BUILTIN(JSM_COMMAND) WITH_ARGS(#JSMCMD) TO_GET(#JS
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

- \* 9. The next step is to issue the READ command to access
- \* the data in the result set. You will notice that this is
- \* done twice in this example, each time accessing
- \* different columns. The capability demonstrated with
- \* this example will be particularly useful in situations
- \* where your result list is larger than what a working
- \* list can hold, so by reducing the number of columns in
- \* the READ you can access the data in more manageable
- \* chunks.
- \* You will note that the first READ has a keyword
- \* SCROLL(\*NO). This allows the second READ to select the
- \* columns from the same records selected by the first
- \* READ. The second READ has a keyword SCROLL(\*YES), which
- \* means that when we begin the next DOWHILE loop, the
- \* first READ will begin with the next set of records. The
- \* number of records read with each READ depends on the

- \* size of the working lists, which in this example have
- \* been set to 100 - NB ENTRYS(100).
- \* Notice that the READ commands include a
- \* SERVICE\_LIST - the fields specified with this keyword
- \* must be the same as the fields defined in the working
- \* list specified in the TO\_GET section of the JSM
- \* command.

```
DOWHILE COND('#JSMSTS *EQ OK')
```

```
CLR_LIST NAMED(#WRKLST1)
CLR_LIST NAMED(#WRKLST2)
CLR_LIST NAMED(#BRWLST1)
CLR_LIST NAMED(#BRWLST2)
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('READ SERVICE_LIST(C
```

```
IF COND('#JSMSTS *EQ NOROW')
LEAVE
ENDIF
```

```
SELECTLIST NAMED(#WRKLST1)
ADD_ENTRY TO_LIST(#BRWLST1)
ENDSELECT
DISPLAY BROWSELIST(#BRWLST1)
```

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS('READ SERVICE_LIST(C
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)
```

```
SELECTLIST NAMED(#WRKLST2)
ADD_ENTRY TO_LIST(#BRWLST2)
ENDSELECT
DISPLAY BROWSELIST(#BRWLST2)
```

```
ENDWHILE
```

- \* 10. After displaying the results disconnect the
- \* service then close JSM.

```
USE BUILTIN(JSM_COMMAND) WITH_ARGS(DISCONNECT) TO_GETI
```

EXECUTE SUBROUTINE(CHECK) WITH\_PARMS(#JSMSTS #JSMMSG)

USE BUILTIN(JSM\_CLOSE) TO\_GET(#JSMSTS #JSMMSG)

EXECUTE SUBROUTINE(CHECK) WITH\_PARMS(#JSMSTS #JSMMSG)

\* SUB ROUTINES

SUBROUTINE NAME(CHECK) PARM((#JSMSTS \*RECEIVED) (#JSMMSG

IF COND('#JSMSTS \*NE OK')

DISPLAY FIELDS(#JSMMSG)

USE BUILTIN(JSM\_CLOSE) TO\_GET(#JSMSTS #JSMMSG)

MENU MSGTXT('Java service error has occurred')

ENDIF

ENDROUTINE

## JSMDirect Example

Loads [Open/Close Example](#) Service. The SERVICE\_LOAD command will be modified, as this function is invoked via JSMDirect and will pass all POST data. The JSM\_CLOSE will cause the last JSMRESPONSE byteArray to be sent back to JSMDirect to write to STDOUT.

```
***** Beginning of RDML commands *****
FUNCTION OPTIONS(*DIRECT)

DEFINE FIELD(#JSMSTS) TYPE(*CHAR) LENGTH(20)
DEFINE FIELD(#JSMMSG) TYPE(*CHAR) LENGTH(255)

***** 'Open service'

USE BUILTIN(JSM_OPEN) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** 'JSMDirect will modify the SERVICE_LOAD command
***** and pass the POST data on with the command

USE BUILTIN(JSM_COMMAND) WITH_ARGS('SERVICE_LOAD SE
 AMPLE1') TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** 'Closing the JSM, will cause last JSMRESPONSE byteArray
***** to be sent back to JSMDirect cgi-bin program'

USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
EXECUTE SUBROUTINE(CHECK) WITH_PARMS(#JSMSTS #JSMMSG)

***** SUB ROUTINES

SUBROUTINE NAME(CHECK) PARMS((#JSMSTS *RECEIVED) (#JSMMSG
 D))

IF COND('#JSMSTS *NE OK')

```

```
***** 'Closing the JSM, will cause any JSMRESPONSE byteArray
***** to be sent back to JSMDirect cgi-bin program'
```

```

```

```
USE BUILTIN(JSM_CLOSE) TO_GET(#JSMSTS #JSMMSG)
```

```

```

```
RETURN
```

```

```

```
ENDIF
```

```

```

```
ENDROUTINE
```

## **RDMLX Examples**

[FTPService Example](#)

[SMTPMailService](#)

[POP3MailService Example](#)

[SMSService Example](#)

[XMLBindFileService Example](#)

[OpenLDAPService Example](#)

[ZipService Example](#)

[PDFSpoolFileService Example](#)

[SVFileService Example](#)

[ExcelReadService Example](#)

[SQLService Example](#)

[aXesTerminalService Example](#)

## FTPService Example

This form connects to a nominated remote FTP server. Once the FTP connection is established, files can be retrieved from the remote server using the GET command or files can be copied from the local server onto the remote server using the PUT command. The current directory on the remote server can be changed as required.

**Modify the default values to be displayed to suit your site.**

\* Uses Integrator Services: FTPService

- \* 1. Allows connection to a remote FTP server
- \* 2. Change the current directory on the remote server
- \* 3. GET a file from or PUT a file onto the remote server

\* The following fields must be defined in your data dictionary to support this function:

|             |       |     |
|-------------|-------|-----|
| * filename  | alpha | 10  |
| * filnbr    | alpha | 6   |
| * jobname   | alpha | 10  |
| * jobuser   | alpha | 10  |
| * jobnbr    | alpha | 6   |
| * status    | alpha | 10  |
| * jsmserver | alpha | 20  |
| * jsmppswrd | alpha | 10  |
| * jsmuserid | alpha | 10  |
| * jsmsts    | alpha | 20  |
| * jsmmsg    | alpha | 255 |
| * jsmpath   | alpha | 150 |
| * jsmfrom   | alpha | 150 |
| * jsmtto    | alpha | 150 |
| * jsmhdlg   | char  | 500 |
| * jsmcmd    | alpha | 4   |

\* Beginning of RDML commands \*\*\*\*\*

function options(\*DIRECT)

```
begin_com role(*EXTENDS #PRIM_FORM) clientheight(260)
clientwidth(615) framestyle(Dialog) height(294) left(459) top(150)
width(623)
```

```
define_com class(#PRIM_GPBX) name(#GPBX_1) caption('Connect to FTP
server') displayposition(1) height(45) left(5) parent(#COM_OWNER)
tabposition(1) tabstop(False) top(5) width(600)
define_com class(#PRIM_GPBX) name(#GPBX_2) caption('FTP details ')
displayposition(2) height(117) left(5) parent(#COM_OWNER) tabposition(2)
tabstop(False) top(60) width(600)
```

```
define_com class(#jsmsserver.Visual) name(#jsmsserver) displayposition(1)
left(8) marginleft(50) parent(#GPBX_1) tabposition(1) top(16) width(209)
define_com class(#jsmuserid.Visual) name(#jsmuserid) displayposition(2)
left(224) marginleft(50) parent(#GPBX_1) tabposition(2) top(16) width(150)
define_com class(#jsmpsswrđ.Visual) name(#jsmpsswrđ) displayposition(3)
left(400) marginleft(50) parent(#GPBX_1) tabposition(3) top(16) width(150)
define_com class(#jsmpath.Visual) name(#jsmpath) caption('Current
Directory') displayposition(1) height(19) labeltype(Caption) left(9)
marginleft(100) parent(#GPBX_2) tabposition(1) top(18) usepicklist(False)
width(555)
define_com class(#jsmfrom.Visual) name(#jsmfrom) displayposition(2)
left(9) marginleft(100) parent(#GPBX_2) tabposition(2) top(47)
define_com class(#jsmto.Visual) name(#jsmto) displayposition(3) left(9)
marginleft(100) parent(#GPBX_2) tabposition(3) top(76)
```

```
define_com class(#PRIM_PHBN) name(#Connect) caption('Connect')
displayposition(1) left(5) parent(#PANL_1) tabposition(1) top(5) width(100)
define_com class(#PRIM_PHBN) name(#Get) caption('Get File')
displayposition(4) enabled(False) left(335) parent(#PANL_1) tabposition(4)
top(5) width(100)
define_com class(#PRIM_PHBN) name(#Put) caption('Put File')
displayposition(3) enabled(False) left(225) parent(#PANL_1) tabposition(3)
top(5) width(100)
define_com class(#PRIM_PHBN) name(#Disconnect) caption('Disconnect')
displayposition(5) enabled(False) left(445) parent(#PANL_1) tabposition(5)
top(5) width(100)
define_com class(#PRIM_PHBN) name(#SetDirectory) caption('Change
Directory') displayposition(2) enabled(False) left(115) parent(#PANL_1)
```

```
tabposition(2) top(5) width(100)
define_com class(#PRIM_STBR) name(#STBR_1) displayposition(4)
height(24) left(0) messageposition(1) parent(#COM_OWNER) tabposition(4)
tabstop(False) top(236) width(615)
define_com class(#PRIM_PANL) name(#PANL_1) displayposition(3)
height(32) left(5) parent(#COM_OWNER) tabposition(3) tabstop(False)
top(187) width(600)
```

```
evtoutine handling(#com_owner.Initialize)
set com(#com_owner) caption(*component_desc)
```

```
#jsmhdl := *default
```

```
* default values
```

```
* #jmsserver := '<server>'
```

```
* #jmsuserid := '<user id>'
```

```
* #jsmpsswr := '<password>'
```

```
#jmsserver := LANSA01
```

```
#jmsuserid := KATE
```

```
#jsmpsswr := LANSA
```

```
endroutine
```

```
mthroutine name(Connect)
```

```
if (#jsmhdl.IsNull)
```

```
* connect the JSMX client to the Java Service Manager and start a thread for
the service
```

```
* Start local JSM server
```

```
use builtin(jsmx_open) to_get(#jmssts #jmsmsg #jsmhdl)
```

```
#com_owner.check(#jmssts #jmsmsg)
```

```
* Load the service
```

```
#jmscmd := 'Service_Load Service(FTPService)'
```

```
use builtin(jsmx_command) with_args(#jsmhdl #jmscmd) to_get(#jmssts
#jmsmsg)
```

```
#com_owner.check(#jsmsts #jsmmsg)
```

```
* Connect to remote FTP server
```

```
#jsmcmd := 'Connect Host(' + #jsmserver + ')
```

```
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts
#jsmmsg)
```

```
#com_owner.check(#jsmsts #jsmmsg)
```

```
* Login
```

```
#jsmcmd := 'Login User(' + #jsmuserid + ') password(' + #jsmpsswrld + ')
```

```
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts
#jsmmsg)
```

```
#com_owner.check(#jsmsts #jsmmsg)
```

```
* get the current directory
```

```
use builtin(jsmx_command) with_args(#jsmhdle GetDir) to_get(#jsmsts
#jsmmsg)
```

```
#jsmpath := #jsmmsg
```

```
endif
```

```
endroutine
```

```
mthroutine name(Disconnect)
```

```
* Quit
```

```
use builtin(jsmx_command) with_args(#jsmhdle 'Quit') to_get(#jsmsts
#jsmmsg)
```

```
#com_owner.check(#jsmsts #jsmmsg)
```

```
* Unload service
```

```
use builtin(jsmx_command) with_args(#jsmhdle 'Service_Unload')
to_get(#jsmsts #jsmmsg)
```

```
#com_owner.check(#jsmsts #jsmmsg)
```

```
* Close the thread
```

```
use builtin(jsmx_close) with_args(#jsmhdle) to_get(#jsmsts #jsmmsg)
```

```
#com_owner.check(#jsmsts #jsmmsg)
```

```
#jsmhdle := *null
```

```
endroutine
```

```
mthroutine name(GetFile)
```

```
* Binary mode
```

```
use builtin(jsmx_command) with_args(#jsmhdle 'Binary') to_get(#jsmsts
#jsmmsg)
```

```
#com_owner.check(#jsmsts #jsmmsg)
```

```
* Get file from remote server
```

```
#jsmcmd := 'get from(' + #jsmfrom + ') to(' + #jsmto + ')'
```

```
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts
#jsmmsg)
```

```
#com_owner.check(#jsmsts #jsmmsg)
```

```
endroutine
```

```
mthroutine name(PutFile)
```

```
* Binary mode
```

```
use builtin(jsmx_command) with_args(#jsmhdle 'Binary') to_get(#jsmsts
#jsmmsg)
```

```
#com_owner.check(#jsmsts #jsmmsg)
```

```
* Put file onto remote server
```

```
#jsmcmd := 'put from(' + #jsmfrom + ') to(' + #jsmto + ')'
```

```
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts
#jsmmsg)
```

```
#com_owner.check(#jsmsts #jsmmsg)
```

```
endroutine
```

```
evtoutine handling(#Connect.Click)
```

```
#com_owner.Connect
```

```
if (#jstmsts = OK)

#Connect.enabled := false
#SetDirectory.enabled := true
#Put.enabled := true
#Get.enabled := true
#Disconnect.enabled := true

endif

endroutine

evtroutine handling(#SetDirectory.Click)

* Change the current directory
#jsmcmd := 'chgdir path(' + #jsmpath + ')'
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jstmsts
#jsmmsg)
#com_owner.check(#jstmsts #jsmmsg)

endroutine

evtroutine handling(#Get.Click)

#com_owner.GetFile

endroutine

evtroutine handling(#Put.Click)

#com_owner.PutFile

endroutine

evtroutine handling(#Disconnect.Click)

#com_owner.DisConnect

if (#jstmsts = OK)
```

```
#Connect.enabled := true
#SetDirectory.enabled := false
#Put.enabled := false
#Get.enabled := false
#Disconnect.enabled := false
```

```
endif
```

```
endroutine
```

```
* check the JSM return status
```

```
mthroutine name(check)
```

```
define_map for(*input) class(#jstmsts) name(#i_status)
```

```
define_map for(*input) class(#prim_alph) name(#i_message)
```

```
message msgtxt(#i_status + ' : ' + #i_message)
```

```
endroutine
```

```
end_com
```

## SMTPMailService

This is a simple function to compose and send an email. This function does not :

Uses Integrator Services: SMTPMAILSERVICE

\* This forms allows you to format and send a simple email using the SMTP protocol.

\* The following fields must be defined in your data dictionary to support this function:

\* jsmemail char 250

\* jsmsts alpha 20

\* jsmsg alpha 255

\* jsmhdl char 500

\* jsmcmd alpha 4

\* Beginning of RDML commands \*\*\*\*\*

function options(\*DIRECT)

begin\_com role(\*EXTENDS #PRIM\_FORM) clientheight(414) clientwidth(58

define\_com class(#PRIM\_STBR) name(#STBR\_1) displayposition(9) height(2

define\_com class(#JSMEMAIL.Visual) name(#toaddress) caption('To') display

define\_com class(#JSMEMAIL.Visual) name(#ccaddress) caption('CC') displa

define\_com class(#JSMEMAIL.Visual) name(#Subject) caption('Subject') disp

define\_com class(#JSMEMAIL.Visual) name(#fromaddress) caption('From Ac

define\_com class(#JSMEMAIL.Visual) name(#fromname) caption('From Nam

define\_com class(#PRIM\_PHBN) name(#Send) caption('Send') displayposition

define\_com class(#PRIM\_PHBN) name(#Reset) caption('Reset') displaypositio

define\_com class(#PRIM\_MEMO) name(#message) componentversion(1) cur

define\_com class(#PRIM\_MECL) name(#messageline) columnrole(Data) disp

define field(#pos) type(\*dec) length(3) decimals(0)

define field(#start) type(\*dec) length(3) decimals(0)

define field(#filename) type(\*char) length(255)

```
def_list name(#filelist) fields(#filename) type(*Working)
def_list name(#tolist) fields(#jsmemail) type(*Working)
def_list name(#cclist) fields(#jsmemail) type(*Working)
```

```
evtroutine handling(#com_owner.Initialize)
set com(#com_owner) caption(*component_desc)
```

```
#jsmhdl := *default
#fromaddress := me@company.com
#fromname := 'LANSA Product Centre'
```

```
* Start JSM Server on IBM i
use builtin(jsmx_open) with_args('ISERIES01:4570') to_get(#jsmsts #jsmmsg
```

```
* Start local JSM server
* use builtin(jsmx_open) with_args('localhost:4560') to_get(#jsmsts #jsmmsg #
* execute subroutine(check) with_parms(#jsmsts #jsmmsg)
```

```
* Load the service
#jsmcmd := 'Service_Load Service(SMTPMailService) trace(*yes)'
use builtin(jsmx_command) with_args(#jsmhdl #jsmcmd) to_get(#jsmsts #jsr
#com_owner.check(#jsmsts #jsmmsg)
```

```
endroutine
```

```
mthroutine name(SendEmail)
```

```
if (*Not #jsmhdl.isnull)
```

```
* from details (ideally these should be set up in your SMTPMailService.proper
```

```
#jsmcmd := 'set from(' + #fromaddress + ') from_name(' + #fromname + ')'
```

```
use builtin(jsmx_command) with_args(#jsmhdl #jsmcmd) to_get(#jsmsts #jsr
#com_owner.check(#jsmsts #jsmmsg)
```

```
* Set the to address(es)
```

```
clr_list named(#tolist)
```

```
#start := 1
```

```
dountil (#pos = 0)
```

```
if (#toaddress.cursize > #start)
#pos := #toaddress.positionof(';', #start)
else
#pos := 0
endif
```

```
if (#pos = 0)
#jsmemail := #toaddress.substring(#start)
else
#jsmemail := #toaddress.substring(#start, (#pos - #start))
#start := #pos + 1
endif
```

```
if (#jsmemail *NE *blank)
add_entry to_list(#tolist)
endif
```

```
enduntil
```

```
use builtin(jsmx_command) with_args(#jsmhdle 'set to(*list) ') to_get(#jsmsts :
#com_owner.check(#jsmsts #jsmmsg)
```

```
* Set the cc address(es)
```

```
clr_list named(#cclist)
```

```
#start := 1
```

```
dountil (#pos = 0)
```

```
if (#ccaddress.cursize > #start)
#pos := #ccaddress.positionof(';', #start)
else
#pos := 0
```

endif

if (#pos = 0)

#jsmemail := #ccaddress.substring( #start )

else

#jsmemail := #ccaddress.substring( #start, (#pos - #start) )

#start := #pos + 1

endif

if (#jsmemail \*NE \*blank)

add\_entry to\_list(#cclist)

endif

enduntil

use builtin(jsmx\_command) with\_args(#jsmhdl 'set cc(\*list)') to\_get(#jsmsts #  
#com\_owner.check( #jsmsts #jsmmsg )

\* add attachments

clr\_list named(#filelist)

#filename := order.xml

add\_entry to\_list(#filelist)

#filename := message01.txt

add\_entry to\_list(#filelist)

#filename := test-input/thankyou.pdf

add\_entry to\_list(#filelist)

\* #jsmcmd := 'add attachment(\*list) zip(orderstatus.zip)'

#jsmcmd := 'add attachment(\*list)'

use builtin(jsmx\_command) with\_args(#jsmhdl #jsmcmd) to\_get(#jsmsts #jsn  
#com\_owner.check( #jsmsts #jsmmsg )

\* Send mail

#jsmcmd := 'send subject(' + #subject + ')'

use builtin(jsmx\_command) with\_args(#jsmhdl #jsmcmd) to\_get(#jsmsts #jsn

```
#com_owner.check(#jsmsts #jsmmsg)

endif

endroutine

* send the email

evtroutine handling(#Send.Click)

#com_owner.SendEmail

endroutine

* reset the email variables

evtroutine handling(#Reset.Click)

clr_list named(#message)

#toaddress #ccaddress #subject := *blank

endroutine

* check the JSM return status

mthroutine name(check)
define_map for(*input) class(#jsmsts) name(#i_status)
define_map for(*input) class(#jsmmsg) name(#i_message)

message msgtxt(#i_status + ' : ' + #i_message)

endroutine

evtroutine handling(#com_owner.closing) options(*noclearmessages *nocleare

* Unload service
use builtin(jsmx_command) with_args(#jsmhdle 'Service_Unload') to_get(#jsn
#com_owner.check(#jsmsts #jsmmsg)
```

```
* Close the thread
use builtin(jsmx_close) with_args(#jsmhdl) to_get(#jsmsts #jsmmsg)
#com_owner.check(#jsmsts #jsmmsg)

endroutine
end_com
```

## POP3MailService Example

This function retrieves all the email messages from the mail server. Details of each message are displayed to the user and, in addition, the body text and any attachments are saved to appropriate directories under the JSM server instance directory.

\* Uses Integrator Services: POP3MailService

\* This forms retrieves messages from a mail server and then deletes them from the mail server.

\*

\* The following fields must be defined in your data dictionary to support this f

```
* jsmserver alpha 20
* jsmppswrd alpha 10
* jsmuserid alpha 10
* jsmfrom alpha 150
* jsmemail char 250
* jsmstring string 1000
* jsmsts alpha 20
* jsmmsg alpha 255
* jsmhdlle char 500
* jsmcmd alpha 4
```

\* Beginning of RDML commands \*\*\*\*\*

```
function options(*DIRECT)
```

```
begin_com role(*EXTENDS #PRIM_FORM) clientheight(414) clientwidth(58
```

```
define_com class(#PRIM_STBR) name(#STBR_1) displayposition(6) height(2
```

```
define_com class(#JSMSEVER.Visual) name(#jsmserver) caption('Server') d
```

```
define_com class(#JSMUSERID.Visual) name(#jsmuser) caption('User') displ
```

```
define_com class(#JSMppswrd.Visual) name(#jsmpassword) caption('Passwor
```

```
define_com class(#PRIM_PHBN) name(#Get) caption('Get Messages') display
```

```
define_com class(#PRIM_PHBN) name(#Reset) caption('Reset') displaypositi
```

```
define_com class(#PRIM_LTVW) name(#message) componentversion(2) disp
```

```
define_com class(#PRIM_LVCL) name(#From) caption('From') captiontype(C
define_com class(#PRIM_LVCL) name(#subject) caption('Subject') captiontyp
define_com class(#PRIM_LVCL) name(#received) caption('Message') caption
```

```
def_list name(#fromlst) fields(#jsmfrom) type(*working)
def_list name(#textlst) fields(#jsmstring) type(*working)
```

```
evtroutine handling(#com_owner.Initialize)
set com(#com_owner) caption(*component_desc)
```

```
#jsmhdle := *default
#jmsserver := '99.99.99.99'
#jmsuser := 'user'
#jmspassword := 'password'
```

```
* Start JSM Server on IBM i
```

```
* use builtin(jsmx_open) with_args('ISERIES01:9990') to_get(#jsmsts #jsmms
```

```
* Start local JSM server
```

```
use builtin(jsmx_open) with_args('localhost:9980') to_get(#jsmsts #jsmmsg #js
#com_owner.check(#jsmsts #jsmmsg)
```

```
* Load the service
```

```
#jsmcmd := 'Service_Load Service(POP3MailService) trace(*yes)'
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsr
#com_owner.check(#jsmsts #jsmmsg)
```

```
endroutine
```

```
mthroutine name(GetEmail)
```

```
if (*Not #jsmhdle.isnull)
```

```
clr_list named(#message)
```

```
* open the post office
```

```
#jsmcmd := 'open server(' + #jmsserver + ') user(' + #jmsuser + ') password(' +
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsr
```

```
#com_owner.check(#jsmsts #jsmmsg)
```

```
* Get count of messages
```

```
use builtin(jsmx_command) with_args(#jsmhdle 'get object(*messagecount)') to
#com_owner.check(#jsmsts #jsmmsg)
```

```
message msgtxt(#jsmsts + ' : ' + #jsmmsg + ' messages in mail box')
```

```
* loop through all the messages
begin_loop
```

```
* Get message
```

```
use builtin(jsmx_command) with_args(#jsmhdle 'get object(*nextmessage)') to
```

```
if (#jsmsts = NOMAIL)
leave
endif
```

```
* Get subject
```

```
use builtin(jsmx_command) with_args(#jsmhdle 'get object(*subject)') to_get(i
#jsmemail := #jsmmsg
```

```
* Get from addresses
```

```
clr_list named(#fromlst)
```

```
use builtin(jsmx_command) with_args(#jsmhdle 'get object(*fromaddress)') to
#com_owner.check(#jsmsts #jsmmsg)
```

```
get_entry number(1) from_list(#fromlst)
```

```
* Read text
```

```
clr_list named(#textlst)
```

```
use builtin(jsmx_command) with_args(#jsmhdle 'read object(*text)') to_get(#js
```

```
#com_owner.check(#jsmsts #jsmmsg)

selectlist named(#textlst)
add_entry to_list(#message)
#jsmemail #jsmfrom := *blank
endselect

* save attachments
use builtin(jsmx_command) with_args(#jsmhdle 'save object(*attachments) dir
#com_owner.check(#jsmsts #jsmmsg)

* text
use builtin(jsmx_command) with_args(#jsmhdle 'save object(*text) file(body1.
#com_owner.check(#jsmsts #jsmmsg)

* Delete message

use builtin(jsmx_command) with_args(#jsmhdle 'delete') to_get(#jsmsts #jsmnr

* Get next message

end_loop

* Close post office

use builtin(jsmx_command) with_args(#jsmhdle 'close') to_get(#jsmsts #jsmm:
#com_owner.check(#jsmsts #jsmmsg)

endif

endroutine

* send the email

evtroutine handling(#Get.Click)

#com_owner.GetEmail

endroutine
```

\* reset the email variables

evtroutine handling(#Reset.Click)

clr\_list named(#message)

endroutine

\* check the JSM return status

mthroutine name(check)

define\_map for(\*input) class(#jsmsts) name(#i\_status)

define\_map for(\*input) class(#jsmmsg) name(#i\_message)

message msgtxt(#i\_status + ' : ' + #i\_message)

endroutine

evtroutine handling(#com\_owner.closing) options(\*noclearmessages \*noclear

\* Unload service

use builtin(jsmx\_command) with\_args(#jsmhdle 'Service\_Unload') to\_get(#jsn  
#com\_owner.check( #jsmsts #jsmmsg )

\* Close the thread

use builtin(jsmx\_close) with\_args(#jsmhdle) to\_get(#jsmsts #jsmmsg)  
#com\_owner.check( #jsmsts #jsmmsg )

endroutine

end\_com

## SMSService Example

This example demonstrates how to use the SMSService in an RDMLX form. The form allows a user to create a message that will be sent to a mobile phone. It consists of a simple interface with 2 input capable fields that need to be defined in the LANSa Repository as:

#PHONE, Alphanumeric, Length 30

#SMSTEXT, Char, Length 200, Visualisation VisualMultiLineEdit

There is also one display only field that needs to be defined in the LANSa Repository as:

#REMCHAR, Integer, Length 3, Decimals 0.

While this example provides a visual interface, it is quite possible that you will build applications that do not need an interface. For example, you might need to automatically send out an SMS when certain conditions are met, like sending an SMS to a sales person when a customer has placed a large order on your web site. Such applications will need to fetch the phone number and message information from a database or some alternative source.

With this example you can enter an email address into the phone number field for initial testing. Once satisfied that it works, you can then start using real phone numbers.



The following is the RDMLX code behind this Visual LANSa form.

```

```

```
*
```

```
* Description:
```

\* This RDMLX form provides an example of how to use the SMSService. The  
 \* It consists of a simple interface with 2 input capable fields that are defined in  
 \* - #PHONE, Alphanumeric, Length 30  
 \* - #SMSTEXT, Char, Length 200, Visualisation VisualMultiLineEdit  
 \* There is also one display only field:  
 \* - #REMCHAR, Integer, Length 3, Decimals 0.

\*  
 \*

\* Disclaimer : The following material is supplied as sample material only. No v  
 \*

\*\*\*\*\*

```
FUNCTION OPTIONS(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(236) Clientwidth
Define_Com Class(#PRIM_LABL) Name(#LABL_1) Alignment(Center) Cap
Define_Com Class(#PHONE.Visual) Name(#PHONE) Displayposition(2) Left
Define_Com Class(#REMCHAR.Visual) Name(#REMCHAR) Displayposition
Define_Com Class(#PRIM_PHBN) Name(#Btn_Send) Caption('Send SMS') L
Define_Com Class(#PRIM_STBR) Name(#STBR_1) Displayposition(5) Heig
Define_Com Class(#SMSTEXT.VisualMultiLineEdit) Name(#SMSTEXT) Dis
```

\* The following locally defined fields are used to hold the parameters required  
 in Functions.

```
Define Field(#JSMSTS) Type(*CHAR) Length(020)
Define Field(#JSMMSG) Type(*CHAR) Length(255)
Define Field(#JSMCMD) Type(*CHAR) Length(255)
Define Field(#JMSRV) Type(*CHAR) Length(050)
Define Field(#JSMHND) Type(*CHAR) Length(4)
```

```
EVTRoutine handling(#com_owner.Initialize)
SET #com_owner caption(*component_desc)
```

\* This example simulates a case whereby the SMS provider can only accept en  
 #Remchar := 200

ENDROUTINE

```
EVTRoutine HANDLING(#Btn_Send.Click)
```

\* 'Open service'

\* The JSMX\_OPEN Builtin Function is used to connect this JSMX client to the  
#JMSRV := 'LANSA01:4560'

Use Builtin(JSMX\_OPEN) With\_Args(#JMSRV) To\_Get(#JMSSTS #JMSMM)  
Execute Subroutine(CHECK) With\_Parms(#JMSSTS #JMSMSG)

\* 'Load service'

\* The Service\_Load(SMSService) command loads and initializes the service  
#JMSCMD := 'Service\_Load Service (SMSService)'

Use Builtin(JSMX\_COMMAND) With\_Args(#JMSHND #JMSCMD) To\_Get  
Execute Subroutine(CHECK) With\_Parms(#JMSSTS #JMSMSG)

\* 'Examples of Set command'

\*

\* Here are two examples of how to use the SET command. This command might

\*

\* 1st Example of 'Set' command. In this example of the 'Set' command, we are  
#JMSCMD := 'Set Encoding(ISO8859\_1)'

Use Builtin(JSMX\_COMMAND) With\_Args(#JMSHND #JMSCMD) To\_Get  
Execute Subroutine(CHECK) With\_Parms(#JMSSTS #JMSMSG)

\* 2nd Example of 'Set' command. In this example of the 'Set' command, we are  
#JMSCMD := 'Set From(return\_address)'

Use Builtin(JSMX\_COMMAND) With\_Args(#JMSHND #JMSCMD) To\_Get  
Execute Subroutine(CHECK) With\_Parms(#JMSSTS #JMSMSG)

\* 'Send message'

\* This command is required to 'kick off' the sending of the message, and contains  
#JMSCMD := 'Send To(' + #Phone + ') Msg(' + #SMSText + ')'

Use Builtin(JSMX\_COMMAND) With\_Args(#JMSHND #JMSCMD) To\_Get  
Execute Subroutine(CHECK) With\_Parms(#JMSSTS #JMSMSG)

\* 'Unload service'

\* This command is required to unload the service and to remove the temporary  
#JMSCMD := 'Service\_Unload'

Use Builtin(JSMX\_COMMAND) With\_Args(#JMSHND #JMSCMD) To\_Get  
Execute Subroutine(CHECK) With\_Parms(#JMSSTS #JMSMSG)

\* 'Close service'

\* The final step in the process is to close the service.

Use Builtin(JSMX\_CLOSE) With\_Args(#JMSHND) To\_Get(#JMSSTS #JMSM)

Execute Subroutine(CHECK) With\_Parms(#JSMSTS #JSMMSG)

Endroutine

\* The following event routine provides information to the user on how many cl  
EVTROUTINE HANDLING(#SMSText.Changed) OPTIONS(\*NOCLEAR

#Remchar := 200 - #SMSText.CurSize

ENDROUTINE

\* SUBROUTINES

\* The following subroutine is used by all the JSMX commands to handle any e  
Subroutine Name(CHECK) Parms((#JSMSTS \*RECEIVED) (#JSMMSG \*RE

If Cond('#JSMSTS \*NE OK')

Message Msgtxt('Java service error has occurred')

Message Msgtxt(#JSMMSG)

Use Builtin(JSMX\_CLOSE) With\_Args(#JSMHND) To\_Get(#JSMSTS #JSM

Endif

Endroutine

END\_COM

## **XMLBindFileService**

### **Fragments Only**

Before you can use this sample code, please read the introduction and perform these steps from the XMLBindFileService RDML Code example:

[XMLBindFileService Example](#)

[Step 1: XML Binding Wizard](#)

[Step 2: Create some folders and some sample XML documents](#)

and then continue to [Step 3. RDMLX Form Code](#) following.

### Step 3. RDMLX Form Code

The following is the RDMLX code that can be used to process these order.  
Please read the notes in the code carefully.

```
* ****
*
* COMPONENT: STD_FORM
*
* ****
```

Function Options(\*DIRECT)

```
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(666) Clientwidth
Define_Com Class(#PRIM_GRID) Name(#ORDXML) Captionnoblanklines(T
Define_Com Class(#PRIM_GDCL) Name(#GDCL_1) Displayposition(1) Pare
Define_Com Class(#PRIM_PHBN) Name(#GETORDERS) Caption('Get Orde
Define_Com Class(#PRIM_PHBN) Name(#PROCESS_ORDER) Caption('Pro
Define_Com Class(#PRIM_GPBX) Name(#GPBX_1) Caption('Order Details')
Define_Com Class(#CUSTNUM.Visual) Name(#CUSTNUM) Displayposition
Define_Com Class(#CUSTNME.Visual) Name(#CUSTNME) Displayposition
Define_Com Class(#ORDDTE.Visual) Name(#ORDDTE) Displayposition(3)
Define_Com Class(#SONUM.Visual) Name(#SONUM) Displayposition(4) He
Define_Com Class(#STREET.Visual) Name(#STREET) Displayposition(5) Le
Define_Com Class(#CITY.Visual) Name(#CITY) Displayposition(6) Left(8) P
Define_Com Class(#STATE.Visual) Name(#STATE) Displayposition(7) Left(8
Define_Com Class(#ZIP.Visual) Name(#ZIP) Displayposition(8) Left(12) Pare
Define_Com Class(#PRIM_GRID) Name(#LINES) Captionnoblanklines(True
Define_Com Class(#PRIM_GDCL) Name(#GDCL_2) Displayposition(1) Pare
Define_Com Class(#PRIM_GDCL) Name(#GDCL_3) Displayposition(2) Pare
Define_Com Class(#PRIM_GDCL) Name(#GDCL_4) Displayposition(3) Pare
Define_Com Class(#PRIM_GDCL) Name(#GDCL_5) Displayposition(4) Pare
Define_Com Class(#PRIM_GDCL) Name(#GDCL_6) Displayposition(5) Pare
Define_Com Class(#PRIM_GRID) Name(#ORDERS) Captionnoblanklines(Ti
Define_Com Class(#PRIM_GDCL) Name(#GDCL_7) Displayposition(1) Pare
Define_Com Class(#PRIM_GDCL) Name(#GDCL_8) Parent(#ORDERS) Sou
Define_Com Class(#PRIM_GDCL) Name(#GDCL_9) Displayposition(2) Pare
Define_Com Class(#PRIM_GDCL) Name(#GDCL_10) Parent(#ORDERS) Sc
Define_Com Class(#PRIM_GDCL) Name(#GDCL_11) Parent(#ORDERS) So
Define_Com Class(#PRIM_GDCL) Name(#GDCL_12) Parent(#ORDERS) Sc
Define_Com Class(#PRIM_GDCL) Name(#GDCL_13) Parent(#ORDERS) Sc
```

Define\_Com Class(#PRIM\_GDCL) Name(#GDCL\_14) Parent(#ORDERS) Sc  
Define\_Com Class(#PRIM\_GDCL) Name(#GDCL\_16) Parent(#LINES) Sour  
Define\_Com Class(#PRIM\_GDCL) Name(#GDCL\_15) Parent(#ORDERS) Sc  
Define\_Com Class(#PRIM\_STBR) Name(#STBR\_1) Displayposition(6) Heig

\* Define the fields to be used in this application

\* Define the fields used by the JSM Commands

Define Field(#JSMSTS) Type(\*CHAR) Length(020)

Define Field(#JSMMSG) Type(\*CHAR) Length(255)

Define Field(#JSMCMD) Type(\*CHAR) Length(255)

Define Field(#JSMHND) Type(\*CHAR) Length(4)

\* NOTE: You will need to define the following commented fields in your repo:

\* These fields will hold values read from and written to the XML documents

\* Define the Order number and date

\* Define Field(#SONUM) Type(\*CHAR) Length(010) Label('Order #')

\* Define Field(#ORDDTE) Type(\*CHAR) Length(010) Label('Order Date')

\*

\* \* Define the Customer details

\* Define Field(#CUSTNUM) Type(\*CHAR) Length(010) Label('Customer #')

\* Define Field(#CUSTNME) Type(\*CHAR) Length(050) Label('Customer name')

\* Define Field(#STREET) Type(\*CHAR) Length(050) Label('Street')

\* Define Field(#CITY) Type(\*CHAR) Length(050) Label('City')

\* Define Field(#STATE) Type(\*CHAR) Length(005) Label('State')

\* Define Field(#ZIP) Type(\*CHAR) Length(005) Label('Post Code')

\* Define the order line details. We will also define a list that holds the order line

\* Define Field(#LINENUM) Type(\*DEC) Length(003) Decimals(0) Label('Line')

\* Define Field(#PARTNUM) Type(\*DEC) Length(003) Decimals(0) Label('Part')

\* Define Field(#PARTDSC) Type(\*CHAR) Length(020) Label('Part Desc.')

\* Define Field(#PARTAMT) Type(\*DEC) Length(010) Decimals(2) Label('Amount')

\* Define Field(#PARTQTY) Type(\*DEC) Length(003) Decimals(0) Label('Quantity')

Define Field(#ORDTOT) Type(\*DEC) Length(010) Decimals(2) Label('Grand Total')

Def\_List Name(#ORDLINES) Fields(#LINENUM #PARTNUM #PARTDSC #PARTAMT #PARTQTY #ORDTOT)

Group\_By Name(#ORDHDR) Fields(#SONUM #ORDDTE #CUSTNUM #CUSTNME #STREET #CITY #STATE #ZIP)

\* The following field will hold the file name and path for the archived files

```
Define Field(#ARCHIVE) Type(*CHAR) Length(30) Desc('Archived Orders I
Define Field(#X_POSN) Type(*DEC) Length(2) Decimals(0) Desc('Working f
```

```
* Define the order response details
```

```
Define Field(#COMMENTS) Type(*CHAR) Length(256) Label('Comments')
Define Field(#LINSTAT) Type(*CHAR) Length(20) Label('Line Status')
Define Field(#RESPONSE) Type(*CHAR) Length(30) Label('Resp filename')
Define Field(#TOGGLE) Type(*DEC) Length(1) Decimals(0)
Def_List Name(#LSTCMNT) Fields(#COMMENTS) Type(*WORKING)
Def_List Name(#RSPLINES) Fields(#LINENUM #PARTNUM #PARTDSC #
```

```
* A single field working list needs to be defined to hold the list of order returns
```

```
* Define Field(#RDXFILENM) Type(*char) Length(100)
Def_List Name(#ORDERLSTW) Fields(#RDXFILENM) Counter(#LISTCOU
```

```
Evtroutine Handling(#com_owner.Initialize)
```

```
Set Com(#com_owner) Caption(*component_desc)
```

```
* We will now start the functionality by opening the JSM
```

```
Use Builtin(JSMX_OPEN) To_Get(#JSMSTS #JSMMSG #JSMHND)
```

```
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
```

```
* Next we will load the JSM service - in this example we have selected to have
```

```
#JSMCMD := 'SERVICE_LOAD SERVICE(XMLBINDFILESERVICE) TRA
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

```
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
```

```
Endroutine
```

```
* SUB ROUTINES
```

```
* The CHECK subroutine is used to capture errors returned from the JSM com
```

```
Subroutine Name(CHECK) Parms((#JSMSTS *RECEIVED) (#JSMMSG *RE
```

```
If ('#JSMSTS *NE OK')
```

```
* Display FIELDS(#JSMSTS #JSMMSG)
```

```
Use Builtin(JSMX_CLOSE) With_Args(#JSMHND) To_Get(#JSMSTS #JSM
```

```
Message Msgtxt('Java service error has occurred')
```

```
Endif
```

Endroutine

Evtroutine Handling(#GETORDERS.Click)

\* Clear out all lists and fields first

Clr\_List Named(#ORDERS)

Clr\_List Named(#LINES)

Clr\_List Named(#ORDERLSTW)

#ORDHDR := \*default

\* The first thing we want to do is to get a list of all

\* the orders that are in the neworders directory. To do

\* this we will use the LIST command. In this scenario,

\* the neworders directory is immediately under the JSM

\* Instance directory on our server, so we can refer to

\* the directory simply as 'neworders'.

\* In this example the file extension has been hard coded

\* as XML. This means that only files with an extension of

\* XML will be returned.

#JSMCMD := 'LIST DIR(NEWORDERS) SERVICE\_LIST(FILENAME) EX'

Use Builtin(JSMX\_COMMAND) With\_Args(#JSMHND #JSMCMD) To\_Get

Execute Subroutine(CHECK) With\_Parms(#JSMSTS #JSMMSG)

\* Display the list returned from the LIST command.

Clr\_List Named(#ORDXML)

Selectlist Named(#ORDERLSTW)

Add\_Entry To\_List(#ORDXML)

Endselect

If ('#LISTCOUNT > 0')

#PROCESS\_ORDER.Enabled := True

Endif

Endroutine

Evtroutine Handling(#PROCESS\_ORDER.Click)

Clr\_List Named(#ORDLINES)

```
#std_num := 0
```

- \* The next step of our application is to run through
- \* the orders in our list and process the data order by
- \* order. To do this we will need to use the READ, BIND,
- \* and GET commands.

```
Selectlist Named(#ORDERLSTW)
```

```
#std_num := #std_num + 1
```

- \* Next we will use the READ command. For the READ
- \* command we need to specify the file that we want
- \* to access, including the file path. In this example,
- \* if an order is named order1.xml for example, then the
- \* FILE keyword would normally be specified as
- \* FILE(neworder/order1.xml), and it would assume that the
- \* neworder directory is under the JSM Instance directory
- \* for the server. We could take this approach, but we
- \* already have the full directory path and file name
- \* specified in the list (in the field #FILENAME) so it
- \* will be much simpler to use this. The actual ARCHIVE
- \* file name and path will be verified in the ARCHIVE
- \* sub-routine.

- \* If we were going to be using a hard coded READ, then
- \* this is what it might look like:

```
* . USE BUILTIN(BUILTIN) WITH_ARGS(JSM_COMMAND)
* . WITH_ARGS('READ FILE(neworder/order1.xml)
* . ARCHIVE(archive/arc_order1.xml)')
* . TO_GET(#JSMSTS #JSMMSG)
```

```
Execute Subroutine(ARCHIVE)
```

```
#JSMCMD := 'READ FILE(' + #RDXFILENM + ') ARCHIVE(' + #ARCHIVI
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
```

- \* The next command to run is the BIND. We named the
- \* service to handle the inbound XML documents as
- \* 'inboundorder'.

```
#JSMCMD := 'BIND SERVICE(INBOUNDORDER) TYPE(*INBOUND) BII
```

Use Builtin(JSMX\_COMMAND) With\_Args(#JSMHND #JSMCMD) To\_Get  
Execute Subroutine(CHECK) With\_Parms(#JSMSTS #JSMMSG)

\* The next step is to retrieve the data using the GET  
\* command.

Use Builtin(JSMX\_COMMAND) With\_Args(#JSMHND 'GET FRAGMENT(  
Execute Subroutine(CHECK) With\_Parms(#JSMSTS #JSMMSG)

#JSMCMD := 'GET FRAGMENT(SALESORDER) SERVICE\_EXCHANGE(  
Use Builtin(JSMX\_COMMAND) With\_Args(#JSMHND #JSMCMD) To\_Get  
Execute Subroutine(CHECK) With\_Parms(#JSMSTS #JSMMSG)

#JSMCMD := 'GET FRAGMENT(CUSTOMER) SERVICE\_EXCHANGE(\*F  
Use Builtin(JSMX\_COMMAND) With\_Args(#JSMHND #JSMCMD) To\_Get  
Execute Subroutine(CHECK) With\_Parms(#JSMSTS #JSMMSG)

#JSMCMD := 'GET FRAGMENT(ORDERDATE) SERVICE\_EXCHANGE(\*  
Use Builtin(JSMX\_COMMAND) With\_Args(#JSMHND #JSMCMD) To\_Get  
Execute Subroutine(CHECK) With\_Parms(#JSMSTS #JSMMSG)

\* The line details of our order technically make up a  
\* list. In this example though, we have defined the lines  
\* as fragments. As such, we need to handle them a little  
\* differently. We need to place the GET FRAGMENT(LINE) in  
\* a loop and continue in the loop until all the line  
\* details are retrieved. The following demonstrates how  
\* we do this.

\* Clr\_List NAMED(#ORDLINES)

Begin\_Loop

Use Builtin(JSMX\_COMMAND) With\_Args(#JSMHND 'GET FRAGMENT(  
If ('#JSMSTS \*EQ NOFRAGMENT')

Leave

Endif

Execute Subroutine(CHECK) With\_Parms(#JSMSTS #JSMMSG)

\* We also need to get the part details for this line.

Use Builtin(JSMX\_COMMAND) With\_Args(#JSMHND 'GET FRAGMENT(  
Add\_Entry To\_List(#ORDLINES)

End\_Loop

- \* Now that we have the data we need and have saved a copy
- \* of the file in the archive, we can close the bind on
- \* this file and delete the file from our inbound
- \* directory.

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND CLOSE) To_Get(#JSMHND #JSMCMD := 'DELETE FILE(' + #RDXFILENM + ')')
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

- \* Add the order header details to the ORDERS grid
- ```
Add_Entry To_List(#ORDERS)
```

- * Now we will work on a response XML in reply to this
- * order. To keep things clean, we will do this in a
- * separate subroutine.

```
Execute Subroutine(RESPONSE)
```

Endselect

- * Clear the header details fields so they do not appear with values until an order
- ```
#ORDHDR := *default
```

- \* Disable the Process Orders button
- ```
#PROCESS_ORDER.Enabled := False
```

Endroutine

- * The ARCHIVE subroutine will build the file name and
- * path for the ARCHIVE keyword of the READ command.

```
Subroutine Name(ARCHIVE)
```

```
Use Builtin(SCANSTRING) With_Args(#RDXFILENM 'neworders' *DEFAULT #X_POSN := #X_POSN + 10
```

```
Substring Field(#RDXFILENM #X_POSN) Into_Field(#ARCHIVE)
```

```
#ARCHIVE := 'archive/arc_' + #ARCHIVE
```

Endroutine

- * This RESPONSE subroutine will do all the processing
- * required to build and create the response XML document.

Subroutine Name(RESPONSE)

- * The first thing we need to do is create a new empty
- * outbound XML document and BIND it to the outbound
- * service that we created with the XML Binding Wizard.
- * Note that we specify the type as *OUTBOUND.

Use Builtin(JSMX_COMMAND) With_Args(#JSMHND 'BIND SERVICE(O
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)

- * Much of the information for the response we already
- * have, but a couple of fields need to be populated so
- * we will make up some fictitious data here. We will
- * create some comments, and add them to a list first.
- * The other new data we need to add is the LINSTAT field
- * to give an indication of the status of the order line.

Clr_List Named(#LSTCMNT)

#COMMENTS := 'Thank you for your order'

Add_Entry To_List(#LSTCMNT)

#COMMENTS := 'We have processed your order ' + #SONUM + ' on date: ' +

Add_Entry To_List(#LSTCMNT)

#COMMENTS := 'Please refer below for the full details of your order.'

Add_Entry To_List(#LSTCMNT)

#COMMENTS := 'One of our sales people will be in touch with you shortly.'

Add_Entry To_List(#LSTCMNT)

#COMMENTS := 'For immediate assistance on your order please call 1234567

Add_Entry To_List(#LSTCMNT)

#COMMENTS := 'one more line'

Add_Entry To_List(#LSTCMNT)

#COMMENTS := 'two more lines'

Add_Entry To_List(#LSTCMNT)

#COMMENTS := 'three more lines'

Add_Entry To_List(#LSTCMNT)

Clr_List Named(#RSPLINES)

- * NOTE: The #TOGGLE field is used to alternate status

- * messages - to add a bit of variety.

Selectlist Named(#ORDLINES)

If (#TOGGLE = 0)

#TOGGLE := 1

```
#LINSTAT := OK
Else
#TOGGLE := 0
#LINSTAT := 'OUT OF STOCK'
Endif
Add_Entry To_List(#RSPLINES)
Endselect
```

* Now that we have some data, we can start using the SET
* command to populate the outbound document object. As
* per the reading of data from the order documents, since
* this example is using fragments only, we will need to
* set up loops to add any repeating data (specifically
* the comments and the order lines).

```
#JSMCMD := 'SET FRAGMENT(ORDERRESPONSE) SERVICE_EXCHAN
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
```

```
Selectlist Named(#LSTCMNT)
```

```
#JSMCMD := 'SET FRAGMENT(RESPONSECOMMENTS) SERVICE_EXC
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
Endselect
```

```
#JSMCMD := 'SET FRAGMENT(SALESORDER) SERVICE_EXCHANGE(
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
```

```
#JSMCMD := 'SET FRAGMENT(CUSTOMER) SERVICE_EXCHANGE(*F
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
```

```
Selectlist Named(#RSPLINES)
```

```
#JSMCMD := 'SET FRAGMENT(LINE) SERVICE_EXCHANGE(*FIELD)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
```

```
#JSMCMD := 'SET FRAGMENT(PART) SERVICE_EXCHANGE(*FIELD)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
```

```
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
Endselect
```

- * Now we have all the data prepared, we will serialize
- * the document object out to a specified file using the
- * WRITE command. We will construct a file name based on
- * the customer and order number.

```
#JSMCMD := 'WRITE FILE(response/RSP_' + #CUSTNUM + '_' + #SONUM
AMOUNT(1)'
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
Execute Subroutine(CHECK) With_Parms(#JSMSTS #JSMMSG)
```

- * Finally we will CLOSE the bind then go onto the next
- * XML document.

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND CLOSE) To_Get(#J
```

```
Endroutine
```

```
Evtoutine Handling(#COM_OWNER.Closing) Options(*NOCLEARMESSA
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND SERVICE_UNLOA
```

```
Use Builtin(JSMX_CLOSE) With_Args(#JSMHND) To_Get(#JSMSTS #JSM
```

```
Endroutine
```

```
Evtoutine Handling(#ORDERS.ItemGotSelection) Options(*NOCLEARMES
```

```
Define Field(#whichone) Reffld(#std_num)
```

```
#whichone := #std_num
```

```
Clr_List Named(#LINES)
```

```
Selectlist Named(#ORDLINES)
```

```
If ('#std_num = #whichone')
```

```
Add_Entry To_List(#LINES)
```

```
Endif
```

```
Endselect
```

```
Endroutine
```

```
End_Com
```

OpenLDAPService Example

This RDMLX form provides an example of how to use the OpenLDAPService. The form allows a user to add the details of a new entry to the IBM Directory Server (IBMTelDir) using the ADD command. It also allows the user to view a list of all current entries using the using the SEARCH command. So, if a new entry is successfully added, this will be reflected in the list.

To run this example you will need to do the following:

1. Configure and start the IBM Directory Server on the IBM i. For more information on the IBM Directory Server please refer to the relevant IBM Manuals.
2. Define two fields in the LANSAs Repository:
 - #JSMSTS, Char, Length 20
 - #JSMMSG, Char, Length 255
3. Set the #JSMSRV field to the correct JSM server value
4. Set the #LDPSRV field to the correct LDAP server value

*

* Description:

* This RDMLX form provides an example of how to use the OpenLDAPService

*

* To run this example you will need to do the following:

* 1. Configure and start the IBM Directory Server on the IBM

i. For more information on the IBM Directory Server please refer to the relevant

* 2. Define two fields in the LANSAs Repository:

* - #JSMSTS, Char, Length 20

* - #JSMMSG, Char, Length 255

* 3. Set the #JSMSRV field to the correct JSM server value

* 4. Set the #LDPSRV field to the correct LDAP server value

*

* Disclaimer : The following material is supplied as sample material only. No v

*

FUNCTION OPTIONS(*DIRECT)

Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(313) Clientwidth

Define_Com Class(#GIVENAME.Visual) Name(#CN_FNAME) Caption('Firs

```
Define_Com Class(#SURNAME.Visual) Name(#SN_LNAME) Caption('Last Name')
Define_Com Class(#STD_DESCCL.Visual) Name(#EMAIL) Caption('Email')
Define_Com Class(#STD_TEXTS.Visual) Name(#PASSWORD) Caption('Password')
Define_Com Class(#PRIM_STBR) Name(#STBR_1) Displayposition(5) Height(20)
Define_Com Class(#PRIM_PHBN) Name(#ADD_RECORD) Caption('Add User')
Define_Com Class(#PRIM_LTVW) Name(#LTVW_1) Componentversion(2)
Define_Com Class(#PRIM_LVCL) Name(#LVCL_1) Caption('Current Directory')
```

* The following locally defined fields are used to hold the parameters required in Functions.

```
Define Field(#JSMCMD) Type(*Char) Length(255)
Define Field(#JSMSRV) Type(*Char) Length(050)
Define Field(#JSMPT) Type(*Char) Length(005)
Define Field(#JSMHND) Type(*Char) Length(4)
Define Field(#LDPSRV) Type(*Char) Length(050)
```

* The following fields and working list are required for the Add command.

```
Define Field(#ATNAME) Type(*Char) Length(050)
Define Field(#ATVALUE) Type(*Char) Length(050)
Def_List Name(#WRKLST) Fields(#ATNAME #ATVALUE) Type(*working)
```

* The following fields and list are used to display the values currently in the display view.

```
Define Field(#DN) Type(*Char) Length(050)
Def_List Name(#WRKLST2) Fields(#DN) Type(*Working)
```

```
EVTROUTINE handling(#com_owner.Initialize)
SET #COM_OWNER caption(*component_desc)
```

* Set up the JSM and LDAP server settings

```
#JSMSRV := '<system-name.><port>'
#LDPSRV := '<ldap-server-name>'
```

* Execute the method to connect to the LDAPServer.

```
#COM_OWNER.Connect
```

* Execute the method to populate the list view.

```
#COM_OWNER.Refresh
```

ENDROUTINE

* Add the record to the directory
EVTROUTINE HANDLING(#ADD_RECORD.Click)

* Execute the Add Method
#COM_OWNER.Add

* Refresh the list view
#COM_OWNER.Refresh

ENDROUTINE

* Add the new entry to the LDAP server
Mthroutine Name(Add)

* The working list that contains the information pertaining to the new record is

#ATNAME := cn
#ATVALUE := #CN_FNAME
Add_Entry To_list(#WRKLST)

#ATNAME := sn
#ATVALUE := #SN_LNAME
Add_Entry To_list(#WRKLST)

#ATNAME := mail
#ATVALUE := #EMAIL
Add_Entry To_list(#WRKLST)

#ATNAME := userPassword
#ATVALUE := #PASSWORD
Add_Entry To_list(#WRKLST)

#ATNAME := objectClass
#ATVALUE := top
Add_Entry To_list(#WRKLST)
#ATVALUE := person
Add_Entry To_list(#WRKLST)
#ATVALUE := inetOrgPerson

```
Add_Entry To_list(#WRKLST)
#ATVALUE := organizationalPerson
Add_Entry To_list(#WRKLST)
```

```
* The Add command is prepared and executed
#JSMCMD := 'Add DN(cn=' + #CN_FNAME + ', cn=users, o=ibmteldir) Serv
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
#COM_OWNER.check(#JSMSTS #JSMMSG)
```

```
ENDROUTINE
```

```
* Refresh the list view
Mthroutine Name(Refresh)
```

```
* 'Search LDAP Server'
```

```
* The SEARCH command is responsible for the actual search of the LDAP dir
#JSMCMD := 'Search Dn(o=ibmteldir) Filter(objectclass=person) Scope(*SUI
Use Builtin(Jsmx_command) With_args(#jsmhnd #jsmcmd) To_Get(#JSMSTS
```

```
* The resultant list is then placed into the list view
Clr_List #Ltvw_1
Selectlist Named(#WRKLST2)
```

```
#STD_DESCL := #DN
```

```
Add_Entry To_List(#Ltvw_1)
```

```
ENDSELECT
```

```
ENDROUTINE
```

```
* The following method routine is used by all the JSMX commands to handle a
Mthroutine name(check)
```

```
define_map for(*input) class(#JSMSTS) name(#I_STATUS)
define_map for(*input) class(#JSMMSG) name(#I_MESSAGE)
```

```
message msgtxt(#JSMSTS.trim + ':' + #JSMMSG)
```

```
endroutine
```

* Connect to the JSM system then the LDAP server
Mthroutine Name(Connect)

* 'Open service'

* The JSMX_OPEN Builtin Function is used to connect this JSMX client to th
Use Builtin(JSMX_OPEN) With_Args(#JSMSRV) To_Get(#JSMSTS #JSM
#COM_OWNER.check(#JSMSTS #JSMMSG)

* 'Load service'

* The Service_Load(OpenLDAPService) command loads and initializes the se
#JSMCMD := 'Service_Load Service (OpenLDAPService)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
#COM_OWNER.check(#JSMSTS #JSMMSG)

* 'Bind to LDAP Server'

* The BIND command is used to establish a connection to the LDAP server. In
#JSMCMD := 'Bind Host(' + #Ldpsrv + ') DN(cn=Administrator) Password(pa
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
#COM_OWNER.check(#JSMSTS #JSMMSG)

Endroutine

*Disconnect from the LDAP server then the JSM system
Mthroutine Name(Disconnect)

* 'Unbind from LDAP Server'

* This command is used to disconnect from the LDAP Server when we have co
#JSMCMD := UnBind
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
#COM_OWNER.check(#JSMSTS #JSMMSG)

* 'Unload service'

* This command is required to unload the service and to remove the temporary
#JSMCMD := 'Service_Unload'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
#COM_OWNER.check(#JSMSTS #JSMMSG)

* 'Close service'

* The final step in the process is to close the service.

Use Builtin(JSMX_CLOSE) With_Args(#JSMHND) To_Get(#JSMSTS #JSM
#COM_OWNER.check(#JSMSTS #JSMMSG)

Endroutine

* When the form is closed we want to disconnect from the LDAP Server

EVTROUTINE HANDLING(#COM_OWNER.Closing) OPTIONS(*NOCLE

#COM_OWNER.Disconnect

ENDROUTINE

END_COM

ZipService Example

This form allows the creation of zip archive files and for existing zip archive files to be opened, reviewed and modified. Zip file contents can also be extracted to a specified directory.

Modify the default values to be displayed to suit your site.

* Uses Integrator Services: ZIPSERVICE

*

* This form is a very simple Windows Zip file processor.

* It allows you to:

* - create zip files

* - open existing zip files and list their contents

* - add files or folders to the open zip file

* - extract the contents of a zip file to a specified folder

*

* NOTE: This program is provided only to demonstrate usage of

* the ZipService. To keep it simple, it has rudimentary error

* handling and does not necessarily demonstrate recommended or

* efficient programming technique or user interface design.

*

* The following field must be defined in your repository:

* filepath string 512

* jsmsts alpha 20

* jsmmsg alpha 255

* jsmhdlg char 500

* jsmcmd alpha 4

*

function options(*DIRECT)

begin_com role(*EXTENDS #PRIM_FORM) bordericons(SystemMenu) capti

define_com class(#PRIM_APPL.ICommonDialogFileOpen) name(#openFileD

define_com class(#PRIM_APPL.ICommonDialogFileSave) name(#saveFileDl

define_com class(#PRIM_LTVW) name(#LVContent) componentversion(2) di

define_com class(#PRIM_LVCL) name(#LVCL_1) displayposition(1) parent(#

```
define_com class(#PRIM_LABL) name(#LABL_1) caption('Zip contents:') display
define_com class(#PRIM_GPBX) name(#GPBX_1) caption('File:') displaypos
define_com class(#PRIM_PHBN) name(#btnNew) caption('&New...') display
define_com class(#PRIM_PHBN) name(#btnOpen) caption('&Open...') display
```

```
define_com class(#PRIM_GPBX) name(#grpAdd) caption('Add:') displayposit
define_com class(#PRIM_PHBN) name(#btnAddFile) caption('Add File...') dis
define_com class(#PRIM_PHBN) name(#btnAddFolder) caption('Add Folder:'
define_com class(#PRIM_LABL) name(#LABL_4) caption('Folder:') display
define_com class(#PRIM_EDIT) name(#edtAddFolder) displayposition(3) hei
define_com class(#PRIM_LABL) name(#lblAddBas) caption('Base:') display
define_com class(#PRIM_CMBX) name(#cboAddBas) comboboxstyle(DropD
define_com class(#PRIM_CBCL) name(#CBCL_1) displayposition(1) parent(
```

```
define_com class(#PRIM_GPBX) name(#grpExtract) caption('Extract:') displa
define_com class(#PRIM_PHBN) name(#btnExtractFile) caption('Extract to:')
define_com class(#PRIM_LABL) name(#LABL_3) caption('Folder:') display
define_com class(#PRIM_EDIT) name(#edtExtractFolder) displayposition(2) l
define_com class(#PRIM_STTC) name(#STTC_1) displayposition(4) height(3
define_com class(#PRIM_LABL) name(#lblExtBas) caption('Base:') displaypc
define_com class(#PRIM_CMBX) name(#cboExtBas) comboboxstyle(DropD
define_com class(#PRIM_CBCL) name(#CBCL_2) displayposition(1) parent(
```

```
define_com class(#PRIM_STBR) name(#STBR_1) displayposition(6) height(2
```

```
define field(#bResult) type(*boolean)
define field(#bIgnore) type(*boolean)
define field(#zippath) type(*string)
define field(#ziptitle) type(*string)
define field(#caption) type(*string)
```

```
def_list name(#ziplist) fields(#filepath) type(*Working) entrys(*MAX)
def_list name(#cmdlist) fields(#std_num #std_name #std_textl #std_text) coun
```

```
evtroutine handling(#com_owner.Initialize)
```

```
set com(#com_owner) caption(*component_desc)
```

```

* start local JSM server and load the ZipService
use builtin(jsmx_open) to_get(#jsmsts #jsmmsg #jsmhandle)
#com_owner.check( #jsmsts #jsmmsg )

#jsmcmd := 'service_load service(zipservice)'
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
#com_owner.check( #jsmsts #jsmmsg )

endroutine

evtroutine handling(#com_owner.Closing)

use builtin(jsmx_command) with_args(#jsmhandle 'service_unload') to_get(#js
#com_owner.check( #jsmsts #jsmmsg )

use builtin(jsmx_close) with_args(#jsmhandle) to_get(#jsmsts #jsmmsg)
#com_owner.check( #jsmsts #jsmmsg )

endroutine

evtroutine handling(#com_owner.CreateInstance) options(*NOCLEARMESS/

* initial state
#com_owner.OnOpenArchive( false )

* load combo boxes
#std_instr #std_inst2 := '*NONE'
add_entry to_list(#cboAddBas)
add_entry to_list(#cboExtBas)

#std_instr #std_inst2 := '*CURRENT'
add_entry to_list(#cboAddBas)
add_entry to_list(#cboExtBas)

#cboAddBas.CurrentItem.Selected := True
#cboExtBas.CurrentItem.Selected := True
#std_instr #std_inst2 := '*PARENT'
add_entry to_list(#cboAddBas)

```

```

* put a hint in the folder edit boxes
#edtAddFolder.Value := '<type folder path>'
#edtExtractFolder.Value := '<type folder path>'

endroutine

* Create an archive file
evtroutine handling(#btnNew.Click)

#sys_appln.CreateFileSaveDialog result(#saveFileDLG)
#saveFileDlg.AddFilter( 'Zip files (*.zip)' '*.zip' )
#saveFileDlg.AddFilter( 'All files (*.*)' '*.*' )
#saveFileDlg.Title := 'New'

#saveFileDlg.Show okpressed(#bResult) formowner(#com_self)

if (#bResult)

* close existing file
#com_owner.Closefile

* create a zip file
#jsmcmd := 'create file(' + #saveFileDlg.file + ')'
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
#com_owner.check( #jsmsts #jsmmsg )

* save details
#zippath := #saveFileDlg.file
#ziptitle := #saveFileDlg.file

#com_owner.OnOpenArchive( true )

#com_owner.caption := #saveFileDlg.file

endif

endroutine

* Select an archive file to open

```

```
evtroutine handling(#btnOpen.Click)
```

```
#sys_appln.CreateFileOpenDialog result(#OpenFileDLG)
```

```
#openFileDlg.AddFilter( 'Zip files (*.zip)' '*.zip' )
```

```
#openFileDlg.AddFilter( 'All files (*.*)' '*.*' )
```

```
#openFileDlg.Show okpressed(#bResult) formowner(#com_self)
```

```
if (#bResult)
```

```
  * open the file
```

```
  #com_owner.OpenFile( #openFileDlg.file #openFileDlg.filetitle )
```

```
  #com_owner.caption := #openFileDlg.file
```

```
endif
```

```
endroutine
```

```
  * select a file to add to the current archive
```

```
evtroutine handling(#btnAddFile.Click)
```

```
#sys_appln.CreateFileOpenDialog result(#OpenFileDLG)
```

```
#openFileDlg.AddFilter( 'All files (*.*)' '*.*' )
```

```
#openFileDlg.Show okpressed(#bResult) formowner(#com_self)
```

```
if (#bResult)
```

```
  * open the current archive file
```

```
  #jsmcmd := 'open file(' + #zippath + ') mode(*write) reaplce(*no)'
```

```
  use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

```
  #com_owner.check( #jsmsts #jsmmsg )
```

```
  * add the specified file to the current archive
```

```
  #jsmcmd := 'add file(' + #openFileDlg.file + ')'
```

```
  use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
```

```
  #com_owner.check( #jsmsts #jsmmsg )
```

```

* close the current archive file
#jsmcmd := 'close'
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
#com_owner.check( #jsmsts #jsmmsg )

* reload archive contents
#com_owner.OpenFile( #zippath #ziptitle )

endif

endroutine

evtroutine handling(#btnAddFolder.Click)

* open the current archive file through the ZipService
#jsmcmd := 'open file(' + #zippath + ') mode(*write) replace(*no)'
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
#com_owner.check( #jsmsts #jsmmsg )

* add the specified folder to the current archive
#jsmcmd := 'add path(' + #edtAddFolder.Value + ') base(' + #cboAddBas.Value
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #
#com_owner.check( #jsmsts #jsmmsg )

* close the current archive through the ZipService
use builtin(jsmx_command) with_args(#jsmhandle 'close') to_get(#jsmsts #jsm
#com_owner.check( #jsmsts #jsmmsg )

* reload archive contents
#com_owner.OpenFile( #zippath #ziptitle )

endroutine

evtroutine handling(#btnExtractFile.Click)

* open the current archive file through the ZipService
#jsmcmd := 'open file(' + #zippath + ') mode(*read)'
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #

```

```
#com_owner.check( #jsmsts #jsmmsg )
```

```
* extract archive contents to the specified folder
```

```
#jsmcmd := 'get entry(*read) to(' + #edtExtractFolder.Value + ') base(' + #cboE  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #  
#com_owner.check( #jsmsts #jsmmsg )
```

```
* close the file
```

```
#jsmcmd := 'close'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #  
#com_owner.check( #jsmsts #jsmmsg )
```

```
endroutine
```

```
* open or create a zip archive, or refresh the displayed contents of the current a  
mthroutine name(OpenFile)
```

```
define_map for(*input) class(#prim_alph) name(#iZipPath)  
define_map for(*input) class(#prim_alph) name(#iZipTitle)
```

```
* close existing file
```

```
#com_owner.Closefile
```

```
* open the file through the ZipService
```

```
#jsmcmd := 'open file(' + #iZipPath + ') mode(*read)'  
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #  
#com_owner.check( #jsmsts #jsmmsg )
```

```
* read the contents into the list box
```

```
clr_list named(#ziplist)
```

```
#jsmcmd := 'get entry(*list)'
```

```
use builtin(jsmx_command) with_args(#jsmhandle #jsmcmd) to_get(#jsmsts #  
#com_owner.check( #jsmsts #jsmmsg )
```

```
selectlist named(#ziplist)
```

```
add_entry to_list(#LVContent)
```

```
endselect
```

```
* save details
#zippath := #iZipPath.Value
#ziptitle := #iZipTitle.Value

#com_owner.OnOpenArchive( true )

endroutine

* Close the current archive
mthroutine name(CloseFile)

clr_list named(#LVContent)

#zippath := *null
#com_owner.OnOpenArchive( false )

endroutine

* Set controls on open or close of archive
mthroutine name(OnOpenArchive)
define_map for(*input) class(#prim_boln) name(#iOpen)

#grpAdd.Enabled := #iOpen
#grpExtract.Enabled := #iOpen

endroutine

* check the JSM return status
mthroutine name(check)
define_map for(*input) class(#jsmsts) name(#i_status)
define_map for(*input) class(#jsmmsg) name(#i_message)

message msgtxt(#i_status + ' : ' + #i_message)

endroutine
end_com
```


PDFSpoolFileService Example

This form opens the default JSM Instance and then explicitly connects to an IBM i to retrieve the relevant spool file information. Refer to commented code for an alternative approach, namely to explicitly OPEN the JSM Server on IBM i and then use an unqualified CONNECT statement to connect to the same machine.

Modify the default values to be displayed to suit your site.

* Uses Integrator Services: PDFSpoolFileService

* This form connects to an IBM

i server to retrieve detail of the spool files in a nominated outq.

* A spool file can then be selected to convert to a PDF file. This form also del
*

* The following fields must be defined in your data dictionary to support this f

* filename	*CHAR	10
* filnbr	*CHAR	6
* jobname	*CHAR	10
* jobuser	*CHAR	10
* jobnbr	*CHAR	6
* status	*CHAR	10
* jsmsserver	*CHAR	10
* jsmpsswrđ	*CHAR	10
* jsmuserid	*CHAR	10
* jsmoutq	*CHAR	10
* jsmsts	alpha	20
* jsmmsg	alpha	255
* jsmhdle	char	500
* jsmcmd	alpha	4

function options(*DIRECT)

begin_com role(*EXTENDS #PRIM_FORM) clientheight(573) clientwidth(69

define_com class(#jsmsserver.Visual) name(#jsmsserver) displayposition(1) heig

define_com class(#jsmuserid.Visual) name(#jsmuserid) displayposition(2) left

define_com class(#jsmpsswrđ.Visual) name(#jsmpsswrđ) displayposition(3) le

define_com class(#jsmoutq.Visual) name(#jsmoutq) displayposition(4) height(

```

define_com class(#jsmpdfdoc.Visual) name(#jsmpdfdoc) displayposition(1) le

define_com class(#PRIM_PHBN) name(#GetSpoolFiles) caption('Connect + C
define_com class(#PRIM_PHBN) name(#ConvertPDF) caption('Convert to PL
define_com class(#PRIM_PHBN) name(#Disconnect) caption('Disconnect') di
define_com class(#PRIM_PHBN) name(#DeletePDF) caption('Delete PDF') di

define_com class(#PRIM_STBR) name(#stbr_1) displayposition(1) height(24)

define_com class(#PRIM_GPBX) name(#GPBX_1) caption('Connect to ') disp
define_com class(#PRIM_GPBX) name(#GPBX_2) caption('PDF file details ')

define_com class(#PRIM_PANL) name(#PANL_1) displayposition(4) height(1
define_com class(#PRIM_PANL) name(#PANL_2) displayposition(2) height(3

define_com class(#PRIM_LTVW) name(#LTVW_1) componentversion(1) disp
define_com class(#PRIM_LVCL) name(#LvCL_1) componenttag('File Name')
define_com class(#PRIM_LVCL) name(#LvCL_2) componenttag('File Numbe
define_com class(#PRIM_LVCL) name(#LvCL_3) componenttag('Job Name')
define_com class(#PRIM_LVCL) name(#LvCL_4) componenttag('Job User') c
define_com class(#PRIM_LVCL) name(#LvCL_5) componenttag('Job Numbe
define_com class(#PRIM_LVCL) name(#LvCL_6) componenttag('Status') disp

def_list name(#spoolst) fields(#filename #filnbr #jobname #jobuser #jobnbr #st

evtroutine handling(#com_owner.Initialize)

set com(#com_owner) caption(*component_desc)

#jsmhdle := *default

* default values
#jsmsserver := '<server>'
#jsmuserid := '<user id>'
#jsmpsswrđ := '<password>'
#jsmoutq := '<outq>'

#jsmpdfdoc := mypdfdoc01.pdf

```

```
#jmsserver.SetFocus
```

```
endroutine
```

```
mthroutine name(Connect)
```

```
if (#jsmhdle.IsNull)
```

```
* connect the JSMX client to the Java Service Manager and start a thread for th
```

```
* Start JSM Server on IBM i
```

```
* use builtin(jsmx_open) with_args('LANSA04:4570') to_get(#jstmsts #jstmmsg
```

```
* Start local JSM server
```

```
use builtin(jsmx_open) to_get(#jstmsts #jstmmsg #jsmhdle)
```

```
#com_owner.check( #jstmsts #jstmmsg )
```

```
* Load the service
```

```
#jsmcmd := 'service_load service(pdfspoolfileservice)'
```

```
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jstmsts #jsr
```

```
#com_owner.check( #jstmsts #jstmmsg )
```

```
* If connected to IBM i JSM Server can use CONNECT w/o parms
```

```
* #jsmcmd := 'Connect '
```

```
* connect to the IBM i host if running local JSM Server
```

```
#jsmcmd := 'Connect Host(' + #jmsserver + ') User(' + #jmsuserid + ') passwor
```

```
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jstmsts #jsr
```

```
#com_owner.check( #jstmsts #jstmmsg )
```

```
endif
```

```
endroutine
```

```
mthroutine name(ListSpoolFiles)
```

```
* Retrieve all spool files created by the nominated user with a *STD formtype
```

```
* #jsmcmd := 'List Library(qursys) queue(' + #jsmoutq + ') user(' + #jmsuseric
```

```
#jsmcmd := 'List Library(qursys) queue(' + #jsmoutq + ') user(' + #jsmuserid -  
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsn  
#com_owner.check( #jsmsts #jsmmsg )
```

```
clr_list named(#ltvw_1)  
selectlist named(#spoolst)
```

```
add_entry to_list(#ltvw_1)
```

```
endselect
```

```
endroutine
```

```
mthroutine name(Convert)
```

```
* add variables for any additional keywords you need to modify
```

```
#jsmcmd := 'Create document(' + #jsmpdfdoc + ') name(' + #filename + ') numb
```

```
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsn  
#com_owner.check( #jsmsts #jsmmsg )
```

```
endroutine
```

```
mthroutine name>DeletePDFFile)
```

```
* delete tje nominated PDF file
```

```
#jsmcmd := 'Delete File(' + #jsmpdfdoc + ')'
```

```
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsn  
#com_owner.check( #jsmsts #jsmmsg )
```

```
endroutine
```

```
mthroutine name(Disconnect)
```

```
if (#jsmhdle <> *blank)
```

```
* disconnect from IBM i host
```

```
use builtin(jsmx_command) with_args(#jsmhdle disconnect) to_get(#jsmsts #js
```

```
#com_owner.check( #jstmsts #jstmmsg )

* unload the service
use builtin(jsmx_command) with_args(#jsmhdle service_unload) to_get(#jsms
#com_owner.check( #jstmsts #jstmmsg )

* Close the thread
use builtin(jsmx_close) with_args(#jsmhdle) to_get(#jstmsts #jstmmsg)

#jsmhdle := *null

endif

endroutine

* check the JSM return status

mthroutine name(check)
define_map for(*input) class(#jstmsts) name(#i_status)
define_map for(*input) class(#jstmmsg) name(#i_message)

message msgtxt(#i_status + ' : ' + #i_message)

endroutine

evtroutine handling(#GetSpoolFiles.Click)

#com_owner.Connect

if (#jstmsts = OK)

#com_owner.ListSpoolFiles

#getspoolFiles.enabled := false
#ConvertPDF.enabled := true
#disconnect.enabled := true
#DeletePDF.enabled := true

endif
```

endroutine

evtroutine handling(#ConvertPDF.Click)

#com_owner.Convert

#getspoolFiles.enabled := false

#ConvertPDF.enabled := true

#disconnect.enabled := true

#DeletePDF.enabled := true

endroutine

evtroutine handling(#Disconnect.Click)

#com_owner.DisConnect

#getspoolFiles.enabled := true

#ConvertPDF.enabled := false

#disconnect.enabled := false

#DeletePDF.enabled := false

endroutine

evtroutine handling(#DeletePDF.Click)

#com_owner.DeletePDFFile

#getspoolFiles.enabled := false

#ConvertPDF.enabled := true

#disconnect.enabled := true

#DeletePDF.enabled := true

endroutine

end_com

SVFileService Example

This simple form allows data to be read from or written to a separated variable file.

- * Uses Integrator Services: SVFileService
- * This forms connects to the local JSM Server and allows SV files to be read, v
- * or written, based on the information currently in the list.
- *
- * The browse and working list definitions are defined with 5 columns to suppo

- * The following field must be defined in your repository:
 - * jsmstring string 1000
 - * jsmsts alpha 20
 - * jsmmsg alpha 255
 - * jsmhdle char 4
 - * jsmcmd alpha 4

```
function options(*DIRECT)
```

```
begin_com role(*EXTENDS #PRIM_FORM) clientheight(573) clientwidth(69
```

```
define_com class(#PRIM_PHBN) name(#Connect) caption('Connect') display
```

```
define_com class(#PRIM_PHBN) name(#Read) caption('Read') displaypositio
```

```
define_com class(#PRIM_PHBN) name(#Disconnect) caption('Disconnect') di
```

```
define_com class(#PRIM_PHBN) name(#Write) caption('Write') displaypositio
```

```
define_com class(#PRIM_PHBN) name(#Clear) caption('Clear') displaypositio
```

```
define_com class(#PRIM_STBR) name(#stbr_1) displayposition(1) height(24)
```

```
define_com class(#PRIM_PANL) name(#PANL_1) displayposition(2) height(1
```

```
define_com class(#PRIM_PANL) name(#PANL_2) displayposition(2) height(3
```

```
define_com class(#PRIM_grid) name(#detailst) columnbuttonheight(15) comp
```

```
define_com class(#PRIM_gdcl) name(#linenum) caption('Line') captiontype(C
```

```
define_com class(#PRIM_gdCL) name(#partnum) caption('Part') captiontype((
```

```
define_com class(#PRIM_gdCL) name(#partdsc) caption('Description') captio
```

```
define_com class(#PRIM_gdCL) name(#partamt) caption('Unit Amount') capti
```

```
define_com class(#PRIM_gdCL) name(#partqty) caption('Quantity') captio
```

```
define_com class(#JSMSTRING.Visual) name(#jsmfile) caption('Path and File
```

```
define_com class(#PRIM_GPBX) name(#GPBX_1) caption('SV type') display
```

```
define_com class(#PRIM_RDBN) name(#CSV) buttonchecked(True) caption(
```

```
define_com class(#PRIM_RDBN) name(#TSV) caption('TSV') displayposition
```

```
define_com class(#PRIM_RDBN) name(#SV) caption('Other') displayposition
```

```
define_com class(#STD_descs.Visual) name(#separator) displayposition(4) hei
```

```
def_list name(#valueslst) fields(#std_num #std_obj #std_desc #std_amnt #std_
```

```
evtroutine handling(#com_owner.Initialize)
```

```
#jsmhdl := *default
```

```
#com_owner.clearlists
```

```
#com_owner.setlist
```

```
endroutine
```

```
evtroutine handling(#Connect.Click)
```

```
#com_owner.Connect
```

```
endroutine
```

```
evtroutine handling(#Read.Click)
```

```
#com_owner.ReadSVFile
```

```
endroutine
```

```
evtroutine handling(#Disconnect.Click)
```

```
#com_owner.DisConnect
```

```
endroutine
```

```
evtroutine handling(#Write.Click)
```

```
#com_owner.WriteSVFile
```

```
endroutine
```

```
evtroutine handling(#Clear.Click)
```

```
#com_owner.ClearLists
```

```
#com_owner.SetList
```

```
endroutine
```

```
mthroutine name(Connect)
```

```
if (#jsmhdl.IsNull)
```

```
* connect the JSMX client to the Java Service Manager and start a thread for th
```

```
* Start JSM Server on IBM i
```

```
* use builtin(jsmx_open) with_args('ISERIES01:9990') to_get(#jsmsts #jsmms
```

```
* Start default JSM server
```

```
use builtin(jsmx_open) to_get(#jsmsts #jsmmsg #jsmhdl)
```

```
#com_owner.check( #jsmsts #jsmmsg )
```

```
* Load the service
```

```
#jsmcmd := 'service_load service(svfileservice)'
```

```
use builtin(jsmx_command) with_args(#jsmhdl #jsmcmd) to_get(#jsmsts #jsr
```

```
#com_owner.check( #jsmsts #jsmmsg )
```

```
#connect.enabled := false
```

```
#read.enabled := true
```

```
#write.enabled := true
```

```
#clear.enabled := true
```

```
#disconnect.enabled := true
```

```
endif
```

```
endroutine
```

```
mthroutine name(ReadSVFile)
```

```
* add variables for any additional keywords you need to modify
```

```
#jsmcmd := 'read file(' + #jsmfile + ')'
```

```
if (#csv.buttonchecked)
```

```
#jsmcmd := #jsmcmd + ' content(*csv)'
```

```
endif
```

```
if (#tsv.buttonchecked)
```

```
#jsmcmd := #jsmcmd + ' content(*tsv)'
```

```
endif
```

```

if (#sv.buttonchecked)

if (#com_owner.checksv = false)
return
endif

#jsmcmd := #jsmcmd + ' content(*sv) separator(' + #separator + ')'

endif

#com_owner.ClearLists

use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsn
#com_owner.check( #jsmsts #jsmmsg )

selectlist named(#valueslst)
add_entry to_list(#detaillst)
endselect

* add some blank entries to allow for additions to the file
#std_obj #std_desc #std_amnt #std_qty := *null
#std_num := #listcount

begin_loop using(#std_count) to(20)
#std_num := #std_num + 1
add_entry to_list(#detaillst)
end_loop

endroutine

mthroutine name(WriteSVFile)

clr_list named(#valueslst)

selectlist named(#detaillst)

if (#std_obj <> *blank)
add_entry to_list(#valueslst)
endif

```

endselect

* add variables for any additional keywords you need to modify

#jsmcmd := 'write file(' + #jsmfile + ')

if (#csv.buttonchecked)

#jsmcmd := #jsmcmd + ' content(*csv)'

endif

if (#tsv.buttonchecked)

#jsmcmd := #jsmcmd + ' content(*tsv)'

endif

if (#sv.buttonchecked)

if (#com_owner.checksv = false)

return

endif

#jsmcmd := #jsmcmd + ' content(*sv) separator(' + #separator + ')

endif

use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsr
#com_owner.check(#jsmsts #jsmmsg)

endroutine

mthroutine name(ClearLists)

clr_list named(#valueslst)

clr_list named(#detaillst)

endroutine

mthroutine name(SetList)

#std_obj #std_desc #std_amnt #std_qty := *null

```
begin_loop using(#std_num) to(20)
add_entry to_list(#detaillst)
end_loop
```

```
endroutine
```

```
mthroutine name(Disconnect)
```

```
if (#jsmhdle <> *blank)
```

```
  * disconnect from IBM i host
```

```
  use builtin(jsmx_command) with_args(#jsmhdle disconnect) to_get(#jsmsts #j:
#com_owner.check( #jsmsts #jsmmsg )
```

```
  * unload the service
```

```
  use builtin(jsmx_command) with_args(#jsmhdle service_unload) to_get(#jsms
#com_owner.check( #jsmsts #jsmmsg )
```

```
  * Close the thread
```

```
  use builtin(jsmx_close) with_args(#jsmhdle) to_get(#jsmsts #jsmmsg)
```

```
#jsmhdle := *null
```

```
#connect.enabled := true
```

```
#read.enabled := false
```

```
#write.enabled := false
```

```
#clear.enabled := false
```

```
#disconnect.enabled := false
```

```
endif
```

```
endroutine
```

```
  * check the JSM return status
```

```
mthroutine name(check)
```

```
define_map for(*input) class(#jsmsts) name(#i_status)
```

```
define_map for(*input) class(#jsmmsg) name(#i_message)
```

```
message msgtxt(#i_status + ' : ' + #i_message)
```

```
endroutine
```

```
mthroutine name(checkSV)
```

```
define_map for(*result) class(#std_bool) name(#i_state) mandatory('true')
```

```
if (#separator *EQ *blank)
```

```
use builtin(message_box_show) with_args(ok ok ERROR ERROR 'Character 1
```

```
#i_state := false
```

```
endif
```

```
endroutine
```

```
end_com
```

ExcelReadService Example

- * Uses Integrator Services: ExcelReadService
- * This forms connects to the local JSM Server and allows Excel files to be read or written, based on the information currently in the list.
- * The browse and working list definitions are defined with 5 columns to support
- * The following field must be defined in your repository:
 - * jsmstring string 1000
 - * jsmstatus string 20
 - * jsmsts alpha 20
 - * jsmmsg alpha 255
 - * jsmhdle char 4
 - * jsmcmd alpha 255

```
function options(*DIRECT)
```

```
begin_com role(*EXTENDS #PRIM_FORM) clientheight(573) clientwidth(69
```

```
define_com class(#PRIM_PHBN) name(#Get) caption('Get') displayposition(1
```

```
define_com class(#PRIM_PHBN) name(#Read) caption('Read') displaypositio
```

```
define_com class(#PRIM_PHBN) name(#Write) caption('Write') displaypositi
```

```
define_com class(#PRIM_PHBN) name(#Clear) caption('Clear') displaypositic
```

```
define_com class(#PRIM_STBR) name(#stbr_1) displayposition(1) height(24)
```

```
define_com class(#PRIM_PANL) name(#PANL_1) displayposition(2) height(1
```

```
define_com class(#PRIM_PANL) name(#PANL_2) displayposition(2) height(3
```

```
define_com class(#PRIM_grid) name(#detaillst) columnbuttonheight(15) comp
```

```
define_com class(#PRIM_gdcl) name(#linenum) caption('Line') captiontype(C
```

```
define_com class(#PRIM_gdCL) name(#partnum) caption('Part') captiontype((
```

```
define_com class(#PRIM_gdCL) name(#partdsc) caption('Description') captio
```

```
define_com class(#PRIM_gdCL) name(#partamt) caption('Unit Amount') capti
```

```
define_com class(#PRIM_gdCL) name(#partqty) caption('Quantity') captionty
```

```
define_com class(#JSMSTRING.Visual) name(#jsmfile) caption('Path and File
```

```
define_com class(#JSMSTATUS.Visual) name(#jsmsheet) caption('Worksheet')
```

```
def_list name(#valueslst) fields(#std_num #std_obj #std_descl #std_amnt #std_
def_list name(#sheetlst) fields(#std_descl) type(*working)
```

```
evtroutine handling(#com_owner.Initialize)
#jsmhdle := *default
```

```
#com_owner.clearlists
#com_owner.setlist
#com_owner.Load
endroutine
```

```
evtroutine handling(#Get.Click)
#com_owner.GetExcelInfo
endroutine
```

```
evtroutine handling(#Read.Click)
#com_owner.ReadExcelFile
endroutine
```

```
evtroutine handling(#Write.Click)
#com_owner.WriteExcelFile
endroutine
```

```
evtroutine handling(#Clear.Click)
#com_owner.ClearLists
#com_owner.SetList
endroutine
```

```
evtroutine handling(#com_owner.closing)
#com_owner.Unload
endroutine
```

```
mthroutine name(Load)
if (#jsmhdle.IsNull)
```

* connect the JSMX client to the Java Service Manager and start a thread for th

```

* Start default JSM server
use builtin(jsmx_open) to_get(#jsmsts #jsmmsg #jsmhdle)
#com_owner.check( #jsmsts #jsmmsg )

* Load the service
#jsmcmd := 'service_load service(ExcelReadService)'
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsr
#com_owner.check( #jsmsts #jsmmsg )

endif
endroutine

* Get EXCEL document information
mthroutine name(GetExcelInfo)

#com_owner.ClearLists
#std_num := 0

* open the document
#com_owner.OpenDocument( read )

* get details about the EXCEL file
#jsmcmd := 'get object(*sheets)'
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsr
#com_owner.check( #jsmsts #jsmmsg )

if (#jsmsts = OK)

#std_num := #std_num + 1
#std_descl := 'Current document has worksheets: '
add_entry to_list(#detaillst)

selectlist named(#sheetlst)
#std_num := #std_num + 1
add_entry to_list(#detaillst)
endselect

endif

```

```

* get rowcount for nominated worksheet
#jsmcmd := 'get object(*rowcount) sheet(' + #jsmsheet + ')'
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsn
#com_owner.check( #jsmsts #jsmmsg )

if (#jsmsts = OK)

#std_num := #std_num + 1

if (#jsmfile = *blank)
#std_descl := 'Number of rows in worksheet sheet1 is ' + #jsmmsg
else
#std_descl := 'Number of rows in worksheet ' + #jsmsheet + ' is ' + #jsmmsg
endif

add_entry to_list(#detailst)

endif

* close all documents
#com_owner.CloseDocuments

endroutine

* Read information from an EXCEL document
mthroutine name(ReadExcelFile)

* open the document
#com_owner.OpenDocument( read )

#com_owner.ClearLists

* read EXCEL file
#jsmcmd := 'read sheet(' + #jsmsheet + ')'
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsn
#com_owner.check( #jsmsts #jsmmsg )

if (#jsmsts = OK)

```

```

selectlist named(#valueslst)
add_entry to_list(#detaillst)
endselect

endif

* add some blank entries to allow for additions to the file
#std_obj #std_desc1 #std_amnt #std_qty := *null
#std_num := #listcount

begin_loop using(#std_count) to(20)
#std_num := #std_num + 1
add_entry to_list(#detaillst)
end_loop

* close all documents
#com_owner.CloseDocuments

endroutine

* Write list details to the EXCEL document
mthroutine name(WriteExcelFile)

clr_list named(#valueslst)

selectlist named(#detaillst)

if (#std_obj <> *blank)

add_entry to_list(#valueslst)

endif

endselect

* open the document
#com_owner.OpenDocument( write )

* add variables for any additional keywords you need to modify

```

```
#jsmcmd := 'write sheet(' + #jsmsheet + ')'  
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsr  
#com_owner.check( #jsmsts #jsmmsg )
```

```
* close all documents  
#com_owner.CloseDocuments
```

```
endroutine
```

```
* Open an EXCEL document  
mthroutine name(OpenDocument)  
define_map for(*input) class(#prim_alph) name(#i_mode)
```

```
* open the nominated EXCEL file  
#jsmcmd := 'open file(' + #jsmfile + ') mode('* + #i_mode + ')'  
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsr  
#com_owner.check( #jsmsts #jsmmsg )
```

```
endroutine
```

```
* Close all open EXCEL documents  
mthroutine name(CloseDocuments)
```

```
#jsmcmd := 'close'  
use builtin(jsmx_command) with_args(#jsmhdle #jsmcmd) to_get(#jsmsts #jsr  
#com_owner.check( #jsmsts #jsmmsg )
```

```
endroutine
```

```
* Clear all lists used for processing  
mthroutine name(ClearLists)
```

```
#valueslst := *null  
clr_list named(#sheetlst)  
clr_list named(#valueslst)  
clr_list named(#detaillst)
```

```
endroutine
```

```

* set default entries in list
mthroutine name(SetList)

#std_obj #std_desc1 #std_amnt #std_qty := *null

begin_loop using(#std_num) to(20)
add_entry to_list(#detaillst)
end_loop

endroutine

* Unload the service
mthroutine name(Unload)

if (#jsmhdl <> *blank)

* unload the service
use builtin(jsmx_command) with_args(#jsmhdl service_unload) to_get(#jsms
#com_owner.check( #jsmsts #jsmmsg )

* Close the thread
use builtin(jsmx_close) with_args(#jsmhdl) to_get(#jsmsts #jsmmsg)

endif

endroutine

* check the JSM return status
mthroutine name(check)
define_map for(*input) class(#jsmsts) name(#i_status)
define_map for(*input) class(#jsmmsg) name(#i_message)

message msgtxt(#i_status + ' : ' + #i_message)

endroutine

end_com
]

```

SQLService Example

This RDMLX form provides an example of using the SQLService. The form allows you to Connect to a database with a JDBC driver, then to Execute a Query against this database. To view the data the user will run a READ command to download the data from the server and display it in a list view.

Note that in this example connection is to an IBM i database called JSMJDBC (since this is an IBM i example, the database name is a library name). The driver name used in the CONNECT command corresponds to the driver name and path defined in the SQLService properties file. The file being accessed is called TBLNAME and it consists of the fields ID, NAME, AGE, SALARY, DEPT, and GRADE.

To run this example you will need to do the following:

- a. Ensure that you have the JDBC driver for the database installed into the jar directory of your LANSA Integrator instance. Also ensure that the SQLServices.properties file contains the details of this JDBC driver and the database that you are connecting to.
- b. Define two fields in the LANSA Repository:
 - #JSMSTS, Char, Length 20
 - #JSMMSG, Char, Length 255
- c. Set the #JSMSRV field to the correct JSM server (and port) value.

This code example uses the following steps:

1. Define the JSMCOMMAND related fields.
2. Define the fields that will be used to map the table fields to the columns in the results list. The fields defined here should match the length and type of the fields defined in the table being queried.
3. Define the 2 fields required for the list that is used to hold the field column mappings.
4. Define the list that will hold the column mappings. This should be a two column list. - the first column in this list will hold the column name and the second field will hold the field name.
5. Define two working lists using the fields defined in the step 2. These lists will be populated by the READ command from the result list that the query generates. In this example, 2 lists are created to demonstrate how you can

keep going back to the result list to access different columns as and when required. Two browselists are defined that are the same shape as these working lists, and these will be used to display the results on a screen.

6. Set up the JSM server settings.
7. Connect to the JSM system then the SQLService.
8. The JSMX_OPEN Built In Function is used to connect this JSMX client to the Java Services Manager, and to start a thread for the service.
9. The Service_Load(SQLService) command loads and initializes the service.
10. Connect to the JDBC driver.
11. Define the column field mappings. This is done by setting #FIELD to the appropriate column and the #1. COLUMN to the appropriate field name (from the table). Each mapping will be added as a new entry to the #MAPLST working list, and then a SET PARAMETER(*MAP) command will be issued. Notice that a SERVICE_LIST is passed as part of this command - the fields defined here are those used in the mapping list. The mapping list must also be specified in the TO_GET portion of the JSM command.
12. Disconnect from the JDBC driver then the JSM system.
13. When the form is closed, disconnect from the service, unload the temporary directory and close the service.

*

* This RDMLX form provides an example of using the
* SQLService. The form allows you to Connect to a
* database with a JDBC driver, then to Execute a
* Query against this database. To view the data the
* user will run a Read command to download the data
* from the server and display it in a list view.

*

* Note that in this example connection is to a
* IBM i database called JSMJDBC (since this is a
* IBM i example, the database name is a library name).
* The driver name used in the CONNECT command
* corresponds to the driver name and path defined
* in the SQLService properties file. The file being
* accessed is called TBLNAME and it consists of the

* fields ID, NAME, AGE, SALARY, DEPT, and GRADE.

*

* Disclaimer : The following material is supplied as
* sample material only. No warranty concerning this
* material or its use in any way whatsoever is
* expressed or implied.

*

FUNCTION OPTIONS(*DIRECT)

Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(380) Clientwidth
Define_Com Class(#PRIM_STBR) Name(#STBR_1) Displayposition(1) Heig
Define_Com Class(#PRIM_LTVW) Name(#READA) Componentversion(2) L
Define_Com Class(#PRIM_PHBN) Name(#READ1) Caption('Read 1') Displa
Define_Com Class(#PRIM_PHBN) Name(#READ2) Caption('Read 2') Displa
Define_Com Class(#PRIM_LVCL) Name(#LVCL_1) Displayposition(1) Paret

* Define the JSMCOMMAND related fields

Define Field(#JSMCMD) Type(*Char) Length(255)
Define Field(#JSMHND) Type(*Char) Length(4)
Define Field(#JSMSRV) Type(*Char) Length(50)

* Define the fields that will be used to map the table fields to the columns in th

Define Field(#COL1) Type(*Char) Length(010)
Define Field(#COL2) Type(*Char) Length(020)
Define Field(#COL3) Type(*Dec) Length(008) Decimals(0)
Define Field(#COL4) Type(*Dec) Length(012) Decimals(2)

* Define the 2 fields required for the list that is used to hold the field column m

Define Field(#FIELD) Type(*Char) Length(010)
Define Field(#COLUMN) Type(*Char) Length(030)

* Define the list that will hold the column mappings. This should be a two colu
Def_List Name(#MAPLST) Fields(#FIELD #COLUMN) Type(*Working)

* Define 2 working lists using the fields defined in the step 2. These lists will b

Def_List Name(#WRKLST1) Fields(#COL1 #COL3) Type(*Working)
Def_List Name(#WRKLST2) Fields(#COL1 #COL2 #COL4) Type(*Working)

EVTROUTINE handling(#com_owner.Initialize)

SET #com_owner caption(*component_desc)

* Set up the JSM server settings

#JSMSRV := 'LANSA01:4560'

* Execute the method to connect to the SQLService.

#COM_OWNER.Connect

* Set up the mapping

#COM_OWNER.Mapping

* Execute the query

#COM_OWNER.RunQuery

ENDROUTINE

* Connect to the JSM system then the SQLService

Mthroutine Name(Connect)

* 'Open service'

* The JSMX_OPEN Builtin Function is used to connect this JSMX client to the
Use Builtin(JSMX_OPEN) With_Args(#JSMSRV) To_Get(#JSMSTS #JSMMSG)
#COM_OWNER.Check(#JSMSTS #JSMMSG)

* 'Load service'

* The Service_Load(SQLService) command loads and initializes the service.

#JSMCMD := 'Service_Load Service (SQLService)'

Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
#COM_OWNER.Check(#JSMSTS #JSMMSG)

* Connect to the JDBC driver

#JSMCMD := 'Connect Driver(DB2) Database(JSMJDBC) User(ALICK) Pass
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
#COM_OWNER.Check(#JSMSTS #JSMMSG)

Endroutine

Mthroutine Name(Mapping)

```
* Define the column field mappings. This is done by setting #FIELD to the appropriate column name
#FIELD := COL1
#COLUMN := ID
Add_Entry To_List(#MAPLST)
#FIELD := COL2
#COLUMN := NAME
Add_Entry To_List(#MAPLST)
#FIELD := COL3
#COLUMN := AGE
Add_Entry To_List(#MAPLST)
#FIELD := COL4
#COLUMN := SALARY
Add_Entry To_List(#MAPLST)
```

```
#JSMCMD := 'Set Parameter(*Map) Service_List(FIELD,COLUMN)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
#COM_OWNER.Check(#JSMSTS #JSMMSG)
```

ENDROUTINE

Mthroutine Name(RunQuery)

```
#JSMCMD := 'Execute Query(Select ID,NAME,AGE,SALARY From TBLNAME)'
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
#COM_OWNER.Check(#JSMSTS #JSMMSG)
```

ENDROUTINE

```
*Disconnect from the JDBC driver then the JSM system
```

Mthroutine Name(Disconnect)

```
* Disconnect from the service
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND DISCONNECT) To_Get
#COM_OWNER.Check(#JSMSTS #JSMMSG)
```

```
* 'Unload service'
```

```
* This command is required to unload the service and to remove the temporary
```

```
#JSMCMD := 'Service_Unload'
```

Use Builtin(JSMX_COMMAND) With_Args(#JSMHND #JSMCMD) To_Get
#COM_OWNER.Check(#JSMSTS #JSMMSG)

* 'Close service'

* The final step in the process is to close the service.

Use Builtin(JSMX_CLOSE) With_Args(#JSMHND) To_Get(#JSMSTS #JSM
#COM_OWNER.Check(#JSMSTS #JSMMSG)

Endroutine

* When the form is closed we want to disconnect from the service

EVTROUTINE HANDLING(#COM_OWNER.Closing) OPTIONS(*NOCL

#COM_OWNER.Disconnect

ENDROUTINE

EVTROUTINE HANDLING(#READ1.Click)

Clr_List #WRKLST1

Clr_List #READA

Use Builtin(JSMX_COMMAND) With_Args(#JSMHND 'Read Service_List(
#COM_OWNER.Check(#JSMSTS #JSMMSG)

Selectlist #wrklst1

#SQLREAD := #COL1.Trim + ' ' + #COL3.AsString.Trim

Add_Entry #reada

EndSelect

ENDROUTINE

EVTROUTINE HANDLING(#READ2.Click)

Clr_List #WRKLST2

Clr_List #READA

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMHND 'Read Service_List((  
#COM_OWNER.Check(#JSMSTS #JSMMSG)
```

```
SelectList #WRKLST2
```

```
#SQLREAD := #COL1.Trim + ' ' + #COL2.Trim + ' ' + #COL4.AsString.Trim  
Add_Entry #READA
```

```
ENDSELECT
```

```
ENDROUTINE
```

```
* The following method routine is used by all the JSMX commands to handle a  
Mthroutine Name(Check)
```

```
Define_Map For(*Input) Class(#JSMSTS) Name(#I_STATUS)  
Define_Map For(*Input) Class(#JSMMSG) Name(#I_MESSAGE)
```

```
Message Msgtxt(#JSMSTS.Trim + ': ' + #JSMMSG)
```

```
Endroutine
```

```
END_COM
```

aXesTerminalService Example

This example will automate a 5250 session, utilising the maintenance programs shipped with the Personnel Demo System, to insert Employee details into the PSLMST file.

In order to demonstrate how to use the Commands and Keywords for the aXesTerminalService, the following sample code written in LANSAs is provided using the JSMX (RDMLX) BIFs.

Note: This example inserts a new employee using the Employee ID A9090. The employee id must be unique in the personnel system files. You must ensure that it does not already exist. Each time you execute the example, you must delete the employee record before executing the example again, or change the employee id used in the code and recompile the function.

Note: This example deliberately uses very limited exception handling code in order that the main program steps might remain clear.

To use this example, you must:

1. Identify the name of a user profile that has access and authority to the LANSAs demonstration personnel system. In addition, the example code as supplied makes the following assumptions:
 - the user's initial menu includes a command entry line at row 18, column 7 (where the example program will type the LANSAs command to start the personnel system employee entry function)
 - the user is authorised to use the commands (LANSAs and SIGNOFF)
 - the user's initial library list includes the necessary LANSAs libraries.

If any of these assumptions are not correct for the user you have chosen, you may to change the supplied example code accordingly.

2. Identify the following details necessary to connect to the aXes Terminal Server on your system:

HOST	The host name of the aXes TerminalServer in your environment. You can usually use the network name of the IBM i system on which aXes is installed and running.
PORT	The port number on which the aXes Terminal Server is listening. By default aXes listens on port 80.
USER	The name of the IBM i user profile defined on the HOST

system that you identified above

PASSWORD The password for the user profile specified by the USER keyword.

3. Change the code in the example that builds and executes the CONNECT service command to use the details you identified above.

```
* -----  
* Process .....: Tutorial  
* Function .....: AXES001  
* Created on .....: July 13, 2011  
* Description ....: Utilising the AxesTerminalService  
* -----
```

Function Options(*DIRECT)

* If these required fields are already defined in the repository then comment or

```
Define Field(#jstmsts) Type(*char) Length(20)  
Define Field(#jmmmsg) Type(*char) Length(255)  
Define Field(#jmxhld1) Type(*char) Length(4)  
Define Field(#jmxcmd) Type(*char) Length(355)
```

* Field to receive the aXes screen name from the aXesTerminalService
* (not used in this example program, but included for illustration)

```
Define Field(#AXSCREEN) Type(*Char) Length(256)
```

* Dummy working list used to exchange fields with the JSM

```
Def_list Name(#EXCHANGE1) Fields(#AXSCREEN) Type(*Working) Entry
```

```
* -----  
* Program Mainline  
* -----
```

* Open a connection to the JSM

```
Use Builtin(JSMX_OPEN) To_Get(#JSMSTS #JSMMSG #jsmxhdl1)
Execute Subroutine(CHECK_STS) With_Parms(#jsmxhdl1)
```

* Load the aXesTerminalService

```
#JSMXCMD := 'SERVICE_LOAD'
```

```
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD 'SERVICE' 'AXES
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD 'TRACE' '*YES')
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
```

* Connect to the aXes server

* NB: PLEASE CHANGE CONNECTION DETAILS TO SUIT YOUR ENVI

* NOTE THE USE OF THE #EXCHANGE1 DUMMY WORKING LIST TO
* FUNCTION FIELD VALUES WITH THE SERVICE. (THE FIELDS ARE
* USED IN THIS EXAMPLE.) YOU COULD ADD 'SERVICE_EXCHANGI
* TO THE JSM COMMAND STRING FOR SIMILAR EFFECT.
* IN THIS CASE, THIS ALLOWS US TO RECEIVE THE AXES SCREEN I
* OF THE RESULTING 5250 SCREEN INTO THE FUNCTION FIELD #AX

```
#JSMXCMD := CONNECT
```

```
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD HOST 'MYSERVE
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD PORT '80')
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD USER 'MYUSER')
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD PASSWORD 'MY
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD #E
Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)
```

* Send (Press Enter) Command to continue

* (This is in case the "Display Program Messages" display is shown)

* (NOTE AGAIN THE USE OF THE #EXCHANGE1 DUMMY WORKING

```
#JSMXCMD := SEND
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD #E
Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)
```

- * Start LANSAs to execute the Enrol function in PLSYS Process
- * (NOTE AGAIN THE USE OF THE #EXCHANGE1 DUMMY WORKING

#JSMXCMD := SETBYPOS

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '18')

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '7')

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD SENDKEY ENTE

Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE("LANSA REQUEST

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDL1 #JSMXCMD #E

Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

- * Type the Employee ID

- * NB: Employee ID used must NOT be already used

#JSMXCMD := SETBYPOS

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '3')

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')

Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(A9090)') To_Get(#JS

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDL1 #JSMXCMD) T

Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

- * Type the Surname

#JSMXCMD := SETBYPOS

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '4')

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')

Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(Smith)') To_Get(#JS

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDL1 #JSMXCMD) T

Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

- * Type the Given Name

#JSMXCMD := SETBYPOS

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '5')

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')

Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(John)') To_Get(#JSM

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDL1 #JSMXCMD) T

Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

* Type the Street Address

```
#JSMXCMD := SETBYPOS
```

```
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '6')
```

```
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')
```

```
Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(1 Some Street)') To_
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T
```

```
Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)
```

* Type the Suburb

```
#JSMXCMD := SETBYPOS
```

```
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '7')
```

```
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')
```

```
Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(Sometown)') To_Get
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T
```

```
Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)
```

* Type the State and Country

```
#JSMXCMD := SETBYPOS
```

```
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '8')
```

```
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')
```

```
Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(CCC)') To_Get(#JSM
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T
```

```
Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)
```

* Type the Zip code

```
#JSMXCMD := SETBYPOS
```

```
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '9')
```

```
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')
```

```
Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(7111)') To_Get(#JSM
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T
```

```
Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)
```

* Type the Home Phone Number

#JSMXCMD := SETBYPOS

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '10')

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')

Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(1222222)') To_Get(#

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T

Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

* Type the Business phone Number

#JSMXCMD := SETBYPOS

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '11')

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')

Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(133333)') To_Get(#J

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T

Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

* Type the Department Code

#JSMXCMD := SETBYPOS

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '12')

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')

Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(ADM)') To_Get(#JS

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T

Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

* Type the Section Code

#JSMXCMD := SETBYPOS

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '13')

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')

Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(04)') To_Get(#JSMX

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T

Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

* Type the Employee Salary

#JSMXCMD := SETBYPOS

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '14')

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')

Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(31222)') To_Get(#JS
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T
Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

- * Type the Start Date
- * AND press ENTER to submit the details
- * (NOTE AGAIN THE USE OF THE #EXCHANGE1 DUMMY WORKING

#JSMXCMD := SETBYPOS

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD SENDKEY ENTE
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '15')
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '43')
Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE(100311)') To_Get(#J
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD #E
Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

- * Press F3 to return to the main menu
- * (NOTE AGAIN THE USE OF THE #EXCHANGE1 DUMMY WORKING

#JSMXCMD := SEND

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD SENDKEY F3)
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD #E
Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

- * Type the SIGNOFF command and press ENTER

#JSMXCMD := SETBYPOS

Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD ROW '18')
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD COL '7')
Execute Subroutine(KEYWRD) With_Parms(#JSMXCMD SENDKEY ENTE
Use Builtin(TCONCAT) With_Args(#jsmxcmd ' VALUE("SIGNOFF")') To_G
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T
Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

- * Disconnect from the aXes terminal server

#JSMXCMD := DISCONNECT

Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T
Execute Subroutine(CHECK_STS) With_Parms(#JSMXCMD)

* Unload the aXesTerminalService

```
#JSMXCMD := 'SERVICE_UNLOAD'
```

```
Use Builtin(JSMX_COMMAND) With_Args(#JSMXHDLE1 #JSMXCMD) T
```

```
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
```

* Close the connection to the JSM

```
Use Builtin(JSMX_CLOSE) With_Args(#JSMXHDLE1) To_Get(#JSMSTS #
```

```
Execute Subroutine(CHECK_STS) With_Parms(#JSMXHDLE1)
```

Return

```
* -----
```

```
* Subroutine to build JSM commands
```

```
* -----
```

```
Subroutine Name(KEYWRD) Parms((#W_CMDX *BOTH) (#W_KEYWRD :
```

```
Define Field(#W_CMDX) Reffld(#JSMXCMD)
```

```
Define Field(#W_KEYWRD) Reffld(#STD_TEXT)
```

```
Define Field(#W_KEYVAL) Length(255) Reffld(#STD_TEXTL)
```

```
#W_CMDX += ' ' + #W_KEYWRD + '(' + #W_KEYVAL + ')'
```

Endroutine

```
* -----
```

```
* Subroutine to check the JSM status
```

```
* -----
```

```
Subroutine Name(CHECK_STS) Parms(#W_HDLE)
```

```
Define Field(#MSGDTA) Type(*CHAR) Length(132)
```

```
Define Field(#W_HDLE) Type(*CHAR) Length(4)
```

```
If Cond('#JSMSTS *NE OK')
```

```
#MSGDTA := 'Error Status Code: ' + #JSMSTS  
Message Msgid(DCM9899) Msgf(DC@M01) Msgdta(#MSGDTA)  
#MSGDTA := 'Error Message: ' + #JSMMSG  
Message Msgid(DCM9899) Msgf(DC@M01) Msgdta(#MSGDTA)
```

```
Endif
```

```
Endroutine
```

RPG Examples

[FTPService Example](#)

[SMTPMailService Example](#)

[POP3MailService Example](#)

[SMSService Example](#)

[ZipService Example](#)

[PDFSpoolFileService Example](#)

[SVFileService Example](#)

[ExcelReadService Example](#)

[ILE RPG Client JSM API Example](#)

FTPService Example

This example program will perform the following steps:

1. Performs a series of calls necessary to load the FTPService
2. Connects and logs on to the remote FTP server you specify using credentials that you specify (you need to modify the source code)
3. Retrieves the name of the current directory from the remote FTP server and writes it to the joblog
4. Changes the current directory on the remote server to the directory LIFTPIN
5. Sends the shipped order.xml file to the remote server, placing it in the LIFTPIN directory
6. Quits the ftp session, unloads the service and closes the connection to the JSM server.

Note:

- To test this example, you need to create the folder /LIFTPIN in the IFS on the remote server.
- You must specify your own values for the remote server name, user id and password before compiling and running the example. These are all contained in constants near the beginning of the source code.
- You may change the directory and file names used in the example if you wish.

Refer to the comments and code in the example for more information.

1. Create and run the ILE RPG example program

Copy and paste the source provided below into a source file member. Then modify the constant values for server, user id and password as directed above and in the source code.

To create the program, you need to use the CRTRPGMOD and CRTPGM commands. Make sure that you use the parameter values specified in the source member.

```
*****
```

```
* FTP:   example in RPG ILE of using the LANSA Integrator
```

```
*       FTPService
```

```
*
```

```
* Note:  This is an example program containing only
```

* rudimentary exception handling
*
* To create this program you must execute the following commands,
* supplying the indicated parameter values and any others that are
* necessary in your installation:

```
* CRTRPGMOD MODULE(<modlib>/FTP)
* SRCFILE(<srclib>/<srcfil>)
*
* CRTPGM PGM(<pgmlib>/FTP)
* MODULE(<modlib>/FTP)
* BNDSRVPGM(<jsmplib>/DCXS882X)
* ACTGRP(*CALLER)
*****
```

* You MUST replace the value of these constants
* before compiling and running this example

```
d ftpserver    c          '<your server>'
d ftpuser      c          '<user id>'
d ftppassword  c          '<password>'
```

* IFS file and folder names used by this program
* - to try this program you need to create the
* folder /LIFTPIN in your IFS
* - this example assumes the shipped ORDER.XML file
* is present in the JSM instance folder

```
d flrtarget    c          '/LIFTPIN'
d ftpsource    c          'ORDER.XML'
d ftptarget    c          'ORDER.XML'
```

* Declare variables for the JSM calls

```
d jsmsrv       s          50a inz(*blanks)
d jsmsts       s          20a inz(*blanks)
d jsmmmsg      s          255a inz(*blanks)
d jsmcmd       s          255a inz(*blanks)
d bytlength    s          10i 0 inz(*zero)
```

* Completion messages

```
d CompMsg01    c          'JSMOPEN call completed.'
d CompMsg02    c          ' SERVICE_LOAD call completed.'
d CompMsg10    c          ' CONNECT call completed.'
d CompMsg20    c          ' LOGIN call completed.'
d CompMsg30    c          ' GETDIR call completed.'
d CompMsg31    c          ' - current directory on +
d                    remote server is: '
d CompMsg40    c          ' CHGDIR call completed.'
d CompMsg50    c          ' BINARY call completed.'
d CompMsg60    c          ' PUT call completed.'
d CompMsg70    c          ' QUIT call completed.'
d CompMsg98    c          ' SERVICE_UNLOAD call completed.'
d CompMsg99    c          'JSMCLOSE call completed.'
```

* Procedure prototypes

```
d CheckResult  pr
d crjsts       const like(jsmsts)
d crjmsg       const like(jsmmsg)
```

```
d SendMessage  pr
d smText       512a VALUE
d smType       10a VALUE
```

* Prototypes for the JSM calls

/COPY QRPGLSRC,JSM_PROC.H

- * Open a connection to the default JSM server
- * - because the server parameter is blank, details of the default
- * JSM server are obtained from the data area JSMCLTDTA on IBM i
- * or from the file jsmcltda.txt on other supported platforms)

```
c          callp  p_jsmopen(jsmsrv:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg01: '*COMP')
```

* Load the FTPService

* - this example explicitly turns tracing on, overriding the settings in the manager.properties file

```
c      eval    jsmcmd = 'SERVICE_LOAD'
c      + ' SERVICE(FTPSERVICE) TRACE(*YES)'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg02:*COMP')
```

* Connect to FTP server

```
c      eval    jsmcmd = 'CONNECT'
c      + ' HOST(' + %trim(ftpserver) + ')'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg10:*COMP')
```

* Login to FTP server

```
c      eval    jsmcmd = 'LOGIN'
c      + ' USER(' + %trim(ftpuser) + ')'
c      + ' PASSWORD(' + %trim(ftppassword) + ')'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg20:*COMP')
```

* Get the current directory on the remote server

* - the current directory is returned in the jsmsg field

```
c      eval    jsmcmd = 'GETDIR'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg30:*COMP')
```

* ... output the current directory name into the joblog ...

```
c      callp   SendMessage(CompMsg31 + %trim(jsmsg))
```

```
c          :*COMP')
```

* Change the current directory on the remote server

```
c          eval    jsmcmd = 'CHGDIR'  
c          + ' PATH(' + %trim(flrtarget) + ')'  
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)  
c          callp  CheckResult(jsmsts:jsmmsg)  
c          callp  SendMessage(CompMsg40:*COMP')
```

* Set binary mode

```
c          eval    jsmcmd = 'BINARY'  
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)  
c          callp  CheckResult(jsmsts:jsmmsg)  
c          callp  SendMessage(CompMsg50:*COMP')
```

* Put file onto remote server

```
c          eval    jsmcmd = 'PUT'  
c          + ' FROM(' + %trim(ftpsource) + ')'  
c          + ' TO(' + %trim(ftptarget) + ')'  
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)  
c          callp  CheckResult(jsmsts:jsmmsg)  
c          callp  SendMessage(CompMsg60:*COMP')
```

* Quit the FTP session

```
c          eval    jsmcmd = 'QUIT'  
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)  
c          callp  CheckResult(jsmsts:jsmmsg)  
c          callp  SendMessage(CompMsg70:*COMP')
```

* Unload the FTPService

```
c          eval    jsmcmd = 'SERVICE_UNLOAD'  
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)  
c          callp  CheckResult(jsmsts:jsmmsg)  
c          callp  SendMessage(CompMsg98:*COMP')
```

* Close the connection to the JSM server and finish

```
c      callp  p_jsmclose(jsmsts:jsmmsg)
c      callp  CheckResult(jsmsts:jsmmsg)
c      callp  SendMessage(CompMsg99:*COMP')
c      eval   *inlr = *on
c      return
```

* Procedure to check the result of a Java Service Manager call

```
p CheckResult  b
d CheckResult  pi
d crjsts              const like(jsmsts)
d crjmsg           const like(jsmmsg)
d crText          s      512a
d crMsg1          c      const('JSM Status : ')
d crMsg2          c      const('JSM Message: ')
d crMsg3          c      const('JSM Service error has +
d                          occurred')
c      if      crjsts <> 'OK'
c      eval    crText = crMsg1 + crjsts
c      callp   SendMessage(crText:*DIAG')
c      eval    crText = crMsg2 + crjmsg
c      callp   SendMessage(crText:*DIAG')
c      callp   SendMessage(crMsg3:*ESCAPE')
c      endif
p CheckResult  e
```

* Procedure to send a program message

```
p SendMessage  b
d SendMessage  pi
d smText              512a  VALUE
d smMsgT           10a  VALUE
```

```
d smMsgI      s      7a  inz('CPF9897')
d smMsgF      s      20a inz('QCPFMSG *LIBL  ')
d smDtaL      s      10i 0 inz(%size(smText))
d smStkE      s      10a  inz('*')
d smStkC      s      10i 0 inz(1)
d smMsgK      s      4a
d smErrC      s      10i 0 inz(0)
```

```
c      if      smMsgT = '*ESCAPE'
c      eval    smMsgI = 'CPF9898'
c      endif
```

```
c      call    'QMHSNDPM'
c      parm    smMsgI
c      parm    smMsgF
c      parm    smText
c      parm    smDtaL
c      parm    smMsgT
c      parm    smStkE
c      parm    smStkC
c      parm    smMsgK
c      parm    smErrC
```

```
p      e
```

SMTPMailService Example

This example program will perform the following steps:

1. It performs a series of calls necessary to load the SMTPMailService
2. It specifies the SMTP server and the user and password necessary to logon to that server using the SET service command (you need to modify the source code to provide details appropriate for your environment)
3. It specifies the TO and FROM address and SUBJECT line for the example e-mail it will send using the SET service command (you need to modify the source code to provide details appropriate for your environment)
4. It sends the example e-mail using the SEND service command – in this simple example, the body text lines comes from compile-time array data coded in the RPG program, but it could equally well have come from a database or have been received in various formats through another LANSA Integrator service call
5. It unloads the service and closes the connection to the JSM server.

Refer to the comments and code in the example for more information.

There are three steps required to make this application work:

1. Create the structure XML

The call to the SEND command of the SMTPMailService passes a multiple occurrence data structure containing the body text lines that will be sent in the e-mail. For this to work, the LANSA Integrator service needs to know the characteristics of this structure. This is accomplished by supplying an XML file that describes the structure.

For this example, the required XML is supplied below. To install this XML you need to perform the following steps:

- a. Locate the structure folder in the JSM instance folder for your JSM server
- b. Create a file called SMTPMailBody.xml
- c. Edit the file with a text editor and paste into it the xml supplied below

Note that the field names used in the structure XML do not need to match the variable names used in the RPG program. It is their order, types and length that are important – not their names.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<rdml:structure xmlns:rdml="http://www.lansa.com/2000/XML/Function">  
  
  <rdml:field name="BODYTXT" type="A" length="80" />  
  
</rdml:structure>
```

2. Register the structure XML with the JSM Server

The example program refers to the structure XML supplied above with the symbolic name SMTP.MailBodyList by specifying that name in the SERVICE_STRUCTURE keyword of the SEND command.

The JSM Server needs to be given a link between the symbolic name and the actual name and location of the structure XML file created in step 1. To do this you need to perform the following steps:

- a. Locate the system folder in the JSM instance folder for your JSM server
- b. Edit the file structure.properties with a text editor and paste into it the entry supplied below (make sure the new entry is on a line by itself)
- c. Save your changes
- d. Restart or refresh the JSM Server instance (refer to [Java Service Manager Refresh](#)).

```
structure.SMTP.MailBodyList=structure/SMTPMailBody.xml
```

3. Create and run the ILE RPG example program

Copy and paste the source provided below into a source file member. Then modify the constant values for server, user id and password and to and from addresses as directed above and in the source code.

To create the program, you need to use the CRTRPGMOD and CRTPGM commands. Make sure that you use the parameter values specified in the source member.

```
*****  
* SMTP:  example in RPG ILE of using the LANSA Integrator  
*      SMTPMailService  
*  
* Note:  This is an example program containing only  
*      rudimentary exception handling  
*  
* To create this program you must execute the following commands,  
* supplying the indicated parameter values and any others that are
```

* necessary in your installation:

*

* CRTRPGMOD MODULE(<modlib>/SMTP)

* SRCFILE(<srclib>/<srcfil>)

*

* CRTPGM PGM(<pgmlib>/SMTP)

* MODULE(<modlib>/SMTP)

* BNDSRVPGM(<jsmpgmlib>/DCXS882X)

* ACTGRP(*CALLER)

* You MUST replace the value of these constants

* before compiling and running this example

* - for smtp_server, specify the network name or address of
* your SMTP mail server (this might not be an IBM i or
* IBM i server). However, it must be addressable from the
* system where the LANSA Integrator JSM server is running

* - for smtp_user, specify the user name used to login to the
* SMTP server

* - for smtp_password, specify the password for the smtp_user
* specified that is used to login to the SMTP server

* NB: user and password might be case sensitive, depending on
* the SMTP server you are using

d smtp_server c '<your server>'

d smtp_user c '<user id>'

d smtp_password c '<password>'

* - for smtp_to, specify the e-mail address you want to send
* the e-mail to (for example, your own e-mail address)

* - for smtp_from, specify the e-mail address you want the
* e-mail to originate from. Because many SMTP mail servers
* prohibit mail relay, this may need to be an address from the
* e-mail domain that is normally managed by the mail server.

- * You can specify your own address - it does not matter if the
- * FROM address is the same as the TO address.

```
d smtp_to      c          '<to address>'
d smtp_from    c          '<from address>'
```

```
d smtp_subject c          'E-mail generated +
d                by SMTPMailService +
d                example program'
```

- * Declare variables for the JSM calls

```
d jsmsrv      s          50a inz(*blanks)
d jsmsts      s          20a inz(*blanks)
d jsmmsg      s          255a inz(*blanks)
d jsmcmd      s          255a inz(*blanks)
d bytelength  s          10i 0 inz(*zero)
```

- * Declare structure to send body text to the SMTPMailService
- * - in this simple example, the data comes from the compile-time
- * array data, but it could equally well have come from a database
- * or received through another LANSA Integrator service call
- * NB: This MUST match the structure xml provided to the JSM Server!

```
d smtpbody    ds          occurs(smtpocur)
d                based(smtpolistptr)
d bodytext    80a
```

```
d smtpocur    c          const(6)
d smtpsize    c          const(%size(smtpbody))
```

- * Declare the compile-time array that provides the body text
- * for this simple example

```
d smtpdata    s          80a dim(smtpocur) perrcd(1) ctdata
```

- * Completion messages

```
d CompMsg01   c          'JSMOPEN call completed.'
```

```

d CompMsg02    c          ' SERVICE_LOAD call completed.'
d CompMsg10    c          ' SET call completed.'
d CompMsg20    c          ' SEND call completed.'
d CompMsg98    c          ' SERVICE_UNLOAD call completed.'
d CompMsg99    c          'JSMCLOSE call completed.'

```

* Procedure prototypes

```

d CheckResult  pr
d crjsts              const like(jsmsts)
d crjmsg              const like(jsmmsg)

```

```

d SendMessage   pr
d smText        512a  VALUE
d smType        10a   VALUE

```

* Prototypes for the JSM calls

/COPY QRPGLSRC,JSM_PROC.H

* Open a connection to the default JSM server
 * - because the server parameter is blank, details of the default
 * JSM server are obtained from the data area JSMCLDTA on IBM i
 * or from the file jsmcltda.txt on other supported platforms)

```

c          callp  p_jsmopen(jsmsrv:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg01:*COMP')

```

* Load the SMTPMailService
 * - this example explicitly turns tracing on, overriding the
 * settings in the manager.properties file

```

c          eval   jsmcmd = 'SERVICE_LOAD'
c          + ' SERVICE(SMTPMAILSERVICE) TRACE(*YES)'
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg02:*COMP')

```

- * Set the SERVER, USER and PASSWORD necessary to connect to
- * your SMTP mail server
- * - you MUST alter the constant definitions at the beginning of
- * this program to specify values appropriate for your mail server
- * - this assumes the mail server listens on port 25.
- * If your mail server uses a different port then you will need
- * to specify the PORT keyword too

```

c      eval    jsmcmd = 'SET'
c          + ' SERVER(' + %trim(smtp_server) + ')'
c          + ' USER(' + %trim(smtp_user) + ')'
c          + ' PASSWORD(' + %trim(smtp_password) + ')'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg10:*COMP')

```

- * Set TO and FROM addresses and SUBJECT
- * - you MUST alter the constant definitions at the beginning of
- * this program to specify values appropriate for your mail server
- * - these can also be specified directly on the SEND command
- * - you can specify a list of TO addresses, you can also specify
- * single or lists of CC and BCC addresses. To specify multiple
- * addresses you must make separate calls to SET for each of TO
- * CC and BCC, supplying the appropriate list of addresses for each

```

c      eval    jsmcmd = 'SET'
c          + ' TO(' + %trim(smtp_to) + ')'
c          + ' FROM(' + %trim(smtp_from) + ')'
c          + ' SUBJECT(' + %trim(smtp_subject) + ')'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg10:*COMP')

```

- * You may wish to specify file attachments to be sent with the e-mail.
- * You can use the ADD service command to add the attachments. This
- * example does not send attachments.

- * Populate the list of e-mail body text lines to be sent
- * - in this simple example, the data comes from the compile-time

- * array data, but it could equally well have come from a database
- * or received through another LANSA Integrator service call

```
c          eval    smtpplistptr = %addr(smtpdata)
```

* Send the e-mail

- * - this passes the multiple occurrence data structure
- * (smtpbody) containing the body text lines
- * - the structure is described to the SMTPMailService by the
- * structure XML identified by the SERVICE_STRUCTURE keyword - th
- * must be a matching entry in the structure.properties file and a
- * corresponding structure XML file, usually in JSMInstance\Structure
- * folder

- * NOTE: this call uses the JSMCMDX api in order to be able to send
- * variable data (in this case the body text structure/list)

```
c          eval    jsmcmd = 'SEND'
c          + ' SERVICE_STRUCTURE(SMTP.MailBodyList)'
c          + ' COUNT(' + %char(smtpocur) + ')'
c          + ' OCCURS(' + %char(smtpocur) + ')'
c          + ' SIZE(' + %char(smtpsize) + ')'
```

```
c          eval    bytelength = smtpocur * smtpsize
```

```
c          callp   p_jsmcmdx(jsmcmd:smtpbody:bytelength:
c                  jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg20:*COMP')
```

* Unload the SMTPMailService

```
c          eval    jsmcmd = 'SERVICE_UNLOAD'
c          callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg98:*COMP')
```

* Close the connection to the JSM server and finish

```

c      callp  p_jsmclose(jsmsts:jsmmsg)
c      callp  CheckResult(jsmsts:jsmmsg)
c      callp  SendMessage(CompMsg99:*'COMP')
c      eval   *inlr = *on
c      return

```

```

*****
* Procedure to check the result of a Java Service Manager call
*****

```

```

p CheckResult  b
d CheckResult  pi
d crjsts              const like(jsmsts)
d crjmsg           const like(jsmmsg)
d crText          s      512a
d crMsg1          c      const('JSM Status : ')
d crMsg2          c      const('JSM Message: ')
d crMsg3          c      const('JSM Service error has +
d                  occurred')
c      if      crjsts <> 'OK'
c      eval   crText = crMsg1 + crjsts
c      callp  SendMessage(crText:*'DIAG')
c      eval   crText = crMsg2 + crjmsg
c      callp  SendMessage(crText:*'DIAG')
c      callp  SendMessage(crMsg3:*'ESCAPE')
c      endif
p CheckResult  e

```

```

*****
* Procedure to send a program message
*****

```

```

p SendMessage  b
d SendMessage  pi
d smText       512a  VALUE
d smMsgT       10a  VALUE
d smMsgI       s      7a  inz('CPF9897')
d smMsgF       s      20a  inz('QCPFMSG *LIBL  ')
d smDtaL       s      10i 0  inz(%size(smText))

```

```

d smStkE      s      10a  inz('*')
d smStkC      s      10i  0  inz(1)
d smMsgK      s          4a
d smErrC      s      10i  0  inz(0)

c      if      smMsgT = '*ESCAPE'
c      eval    smMsgI = 'CPF9898'
c      endif

c      call    'QMHSNDPM'
c      parm    smMsgI
c      parm    smMsgF
c      parm    smText
c      parm    smDtaL
c      parm    smMsgT
c      parm    smStkE
c      parm    smStkC
c      parm    smMsgK
c      parm    smErrC

```

p e

**CTDATA smtpdata

This e-mail was generated by the LANSA Integrator RPG ILE example program for the SMTPMailService.

Refer to the LANSA Integrator Guide for more information about using the SMTPMailService and other LANSA Integrator services.

=====

<http://www.lansa.com>

POP3MailService Example

This example program will perform the following steps:

1. It performs a series of calls necessary to load the POP3MailService.
2. It opens a user mailbox on the POP3 mail server, using the POP3 server and the user and password details provided (you need to modify the source code to provide details appropriate for your environment).
3. It retrieves a count of messages available in the mailbox. If there is more than one message, it retrieves various details of the first message and writes them to the joblog and then saves the message body text to file body1.txt in folder mailbody in the JSM instance directory.

Note: as supplied the example program does NOT remove the message from the server mailbox, so if you run the example more than once you may receive the same message each time.

4. It closes the mailbox, unloads the service and closes the connection to the JSM server.

Note:

- To test this example fully, you need to be sure there is at least one e-mail message in the mailbox on the POP3 server for the specified user. If necessary, use your e-mail client program to send a test e-mail to the specified user.
- You must specify your own values for the remote server name, user id and password before compiling and running the example. These are all contained in constants near the beginning of the source code.
- You may change the directory and file names used in the example if you wish.

Refer to the comments and code in the example for more information.

Create and run the ILE RPG example program

Copy and paste the source provided below into a source file member. Then modify the constant values for server, user id and password as directed above and in the source code.

To create the program, you need to use the CRTRPGMOD and CRTPGM commands. Make sure that you use the parameter values specified in the source

member.

* POP3: example in RPG ILE of using the LANSA Integrator
* POP3MailService

*

* Note: This is an example program containing only
* rudimentary exception handling

*

* To create this program you must execute the following commands,
* supplying the indicated parameter values and any others that are
* necessary in your installation:

*

* CRTRPGMOD MODULE(<modlib>/POP3)
* SRCFILE(<srclib>/<srcfil>)

*

* CRTPGM PGM(<pgmlib>/POP3)
* MODULE(<modlib>/POP3)
* BNDSRVPGM(<jsmpgmlib>/DCXS882X)
* ACTGRP(*CALLER)

* You MUST replace the value of these constants
* before compiling and running this example

* - for pop3_server, specify the network name or address of
* your POP3 mail server (this might not be an IBM i or
* IBM i server). However, it must be addressable from the
* system where the LANSA Integrator JSM server is running

* - for pop3_user, specify the user name used to login to the
* POP3 server

* - for pop3_password, specify the password for the pop3_user
* specified that is used to login to the POP3 server

* NB: user and password might be case sensitive, depending on
* the POP3 server you are using

```
d pop3_server    c          '<your server>'
d pop3_user      c          '<user id>'
d pop3_password  c          '<password>'
```

- * These constants specify the location that this example program
- * will save the e-mail body text - change them if required
- * - because a relative path is specified, the shipped default
- * will save to folder "mailbody" in the jsm instance folder

```
d pop3_savefil   c          'body1.txt'
d pop3_savedir   c          'mailbody'
```

- * Declare variables for the JSM calls

```
d jsmsrv        s          50a inz(*blanks)
d jsmsts         s          20a inz(*blanks)
d jsmmsg         s          255a inz(*blanks)
d jsmcmd         s          255a inz(*blanks)
d bytelength     s          10i 0 inz(*zero)
```

- * Completion messages

```
d CompMsg01     c          'JSMOPEN call completed.'
d CompMsg02     c          ' SERVICE_LOAD call completed.'
d CompMsg10     c          ' OPEN call completed.'
d CompMsg20     c          ' GET OBJECT(*MESSAGECOUNT) call +
d                completed'
d CompMsg21     c          'Message count is: '
d CompMsg30     c          ' GET OBJECT(*FIRSTMESSAGE) call +
d                completed'
d CompMsg40     c          ' GET OBJECT(*SUBJECT) call +
d                completed'
d CompMsg41     c          'Subject is: '
d CompMsg50     c          ' GET OBJECT(*FROMADDRESS) call +
d                completed'
d CompMsg51     c          'FROM address is: '
d CompMsg60     c          ' GET OBJECT(*SENTDATE) call +
d                completed'
d CompMsg61     c          'Sent date is: '
```

```

d CompMsg70    c          ' SAVE OBJECT(*TEXT) call +
d              completed'
d CompMsg80    c          ' DELETE call completed.'
d CompMsg90    c          ' CLOSE call completed.'
d CompMsg98    c          ' SERVICE_UNLOAD call completed.'
d CompMsg99    c          'JSMCLOSE call completed.'

```

* Procedure prototypes

```

d CheckResult  pr
d crjsts       const like(jsmsts)
d crjmsg       const like(jsmmsg)

```

```

d SendMessage  pr
d smText       512a  VALUE
d smType       10a   VALUE

```

* Prototypes for the JSM calls

/COPY QRPGLSRC,JSM_PROC.H

* Open a connection to the default JSM server
 * - because the server parameter is blank, details of the default
 * JSM server are obtained from the data area JSMCLTDTA on IBM i
 * or from the file jsmcltdta.txt on other supported platforms)

```

c          callp  p_jsmopen(jsmsrv:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg01:*COMP')

```

* Load the Pop3MailService

* - this example explicitly turns tracing on, overriding the
 * settings in the manager.properties file

```

c          eval   jsmcmd = 'SERVICE_LOAD'
c          + ' SERVICE(POP3MAILSERVICE) TRACE(*YES)'
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg02:*COMP')

```

- * Open the post office (login to the POP3 server)
- * - you MUST alter the constant definitions at the beginning of
- * this program to specify values appropriate for your mail server
- * for server, user id and password
- * - this assumes the mail server listens on port 110.
- * If your mail server uses a different port then you will need
- * to specify the PORT keyword too

```

c          eval    jsmcmd = 'OPEN'
c          + ' SERVER(' + %trim(pop3_server) + ')'
c          + ' USER(' + %trim(pop3_user) + ')'
c          + ' PASSWORD(' + %trim(pop3_password) + ')'
c          callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg10:*COMP')

```

- * Get the count of available e-mail messages

```

c          eval    jsmcmd = 'GET OBJECT(*MESSAGECOUNT)'
c          callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg20:*COMP')

```

- * Output the count to the joblog, proceed if one or more messages ...

```

c          callp   SendMessage(CompMsg21 + %trim(jsmmsg)
c                  :*COMP')
c          if      %int(jsmmsg) > 0

```

- * ... set the current message to the first available message ...

```

c          eval    jsmcmd = 'GET OBJECT(*FIRSTMESSAGE)'
c          callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg30:*COMP')

```

- * ... get the message subject and write to joblog ...

```
c      eval    jsmcmd = 'GET OBJECT(*SUBJECT)'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg40:*COMP')

c      callp   SendMessage(CompMsg41 + %trim(jsmmsg)
c              :*COMP')
```

* ... get the FROM address and write to joblog ...

```
c      eval    jsmcmd = 'GET OBJECT(*FROMADDRESS)'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg50:*COMP')

c      callp   SendMessage(CompMsg51 + %trim(jsmmsg)
c              :*COMP')
```

* ... get the sent date and write to joblog ...

```
c      eval    jsmcmd = 'GET OBJECT(*SENTDATE)'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg60:*COMP')

c      callp   SendMessage(CompMsg61 + %trim(jsmmsg)
c              :*COMP')
```

* ... save the body text to a file

* - as supplied, the body text will be saved to file "body1.txt"
* in directory "mailbody" within the jsm instance directory

```
c      eval    jsmcmd = 'SAVE OBJECT(*TEXT)'
c              + ' FILE(' + %trim(pop3_savefil) + ')'
c              + ' DIR(' + %trim(pop3_savedir) + ')'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg70:*COMP')
```

* ... after successfully processing a message, many applications
 * would wish to remove the message from the mailbox on the POP3
 * server. As shipped, this example program does not do that but
 * you can uncomment the following code to do so if you wish ...

```
c*****eval      jsmcmd = 'DELETE'
c*****callp     p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c*****callp     CheckResult(jsmsts:jsmmsg)
c*****callp     SendMessage(CompMsg80:*COMP')
```

```
c      endif
```

* Close the post office

```
c      eval      jsmcmd = 'CLOSE'
c      callp     p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp     CheckResult(jsmsts:jsmmsg)
c      callp     SendMessage(CompMsg90:*COMP')
```

* Unload the Pop3MailService

```
c      eval      jsmcmd = 'SERVICE_UNLOAD'
c      callp     p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp     CheckResult(jsmsts:jsmmsg)
c      callp     SendMessage(CompMsg98:*COMP')
```

* Close the connection to the JSM server and finish

```
c      callp     p_jsmclose(jsmsts:jsmmsg)
c      callp     CheckResult(jsmsts:jsmmsg)
c      callp     SendMessage(CompMsg99:*COMP')
c      eval      *inlr = *on
c      return
```

* Procedure to check the result of a Java Service Manager call

```
p CheckResult  b
d CheckResult  pi
```

```

d crjsts                const like(jsmsts)
d crjmsg                const like(jsmmsg)
d crText                s      512a
d crMsg1                c      const('JSM Status : ')
d crMsg2                c      const('JSM Message: ')
d crMsg3                c      const('JSM Service error has +
d                          occurred')
c                          if    crjsts <> 'OK'
c                          eval  crText = crMsg1 + crjsts
c                          callp SendMessage(crText:'*DIAG')
c                          eval  crText = crMsg2 + crjmsg
c                          callp SendMessage(crText:'*DIAG')
c                          callp SendMessage(crMsg3:'*ESCAPE')
c                          endif
p CheckResult          e

```

* Procedure to send a program message

```

p SendMessage          b

```

```

d SendMessage          pi
d smText                512a VALUE
d smMsgT                10a VALUE

d smMsgI                s      7a  inz('CPF9897')
d smMsgF                s      20a inz('QCPFMSG *LIBL  ')
d smDtaL                s      10i 0 inz(%size(smText))
d smStkE                s      10a  inz('*')
d smStkC                s      10i 0 inz(1)
d smMsgK                s      4a
d smErrC                s      10i 0 inz(0)

c                          if    smMsgT = '*ESCAPE'
c                          eval  smMsgI = 'CPF9898'
c                          endif

c                          call  'QMHSNDPM'
c                          parm          smMsgI

```

c	parm	smMsgF
c	parm	smText
c	parm	smDtaL
c	parm	smMsgT
c	parm	smStkE
c	parm	smStkC
c	parm	smMsgK
c	parm	smErrC

p e

intengbr_RPG_STMP

SMSService Example

```
*****
* SMS:   Example in RPG ILE of using the LANSA Integrator
*        SMSService.
*        This example uses the SMTP (email) protocol to send the
*        SMS details to the SMS gateway.
*        You need to check:
*        - The SMS gateway you are using accepts SMS messages
*          for transmission in an email format.
*        - The format of the email sent agrees with the format
*          expected by your SMS gateway.
*
* Note:  This is an example program containing only
*        rudimentary exception handling
*
* To create this program you must execute the following commands,
* supplying the indicated parameter values and any others that are
* necessary in your installation:
*
* CRTRPGMOD MODULE(<modlib>/SMS)
*          SRCFILE(<srclib>/<srcfil>)
*
* CRTPGM   PGM(<pgmlib>/SMS)
*          MODULE(<modlib>/SMS)
*          BNDSRVPGM(<lansalib>/DCXS882X)
*          ACTGRP(*CALLER)
*****
* To successfully run this example you will need an SMTP (email)
* server and an SMS gateway.
* These need to be set up in the SMSService.properties file.
* For example:
*   transport=smtp
*   port=25
*   server=10.2.0.55
*   subject=user+password
*   from.address=your.name@yourcoy.com.au
```

```
* mail.domain=yourcoy.com.au
* mobile.domain=streetdata.com.au
*****
```

```
* You must replace the value of the following constants
* before compiling and running this example
```

```
* phoneno value needs to be replaced with mobile phone number to
* which the SMS message will be sent.
d phoneno      c          '+61412345678'
```

```
* message value can be replaced with the text of the
* SMS message
d message      c          'Hello. Test message'
```

```
* emailfrom value needs to be replaced with the email addressee
* which will be used on the SMTP (email) sent to the SMS gateway.
d emailfrom    c          'fred.smith@lansa.com.au'
```

```
*****
```

```
*
* Declare variables for the JSM calls
```

```
d jsmsrv      s          50a inz(*blanks)
d jsmsts      s          20a inz(*blanks)
d jsmmsg      s          255a inz(*blanks)
d jsmcmd      s          255a inz(*blanks)
d bytelength  s          10i 0 inz(*zero)
```

```
* Completion messages
```

```
d CompMsg01   c          'JSMOPEN call completed.'
d CompMsg02   c          ' SERVICE_LOAD call completed.'
d CompMsg11   c          ' SET ENCODING call completed.'
d CompMsg12   c          ' SET FROM call completed.'
d CompMsg20   c          ' SEND call completed.'
d CompMsg98   c          ' SERVICE_UNLOAD call completed.'
d CompMsg99   c          'JSMCLOSE call completed.'
```

* Procedure prototypes

```
d CheckResult    pr
d crjsts          const like(jsmsts)
d crjmsg          const like(jsmmsg)

d SendMessage    pr
d smText         512a VALUE
d smType         10a VALUE
```

* Prototypes for the JSM calls

```
/COPY QRPGLSRC,JSM_PROC.H
```

```
*****
```

* Open a connection to the default JSM server
* - because the server parameter is blank, details of the default
* JSM server are obtained from the data area JSMCLTDTA on IBM i
* or from the file jsmcltdta.txt on other supported platforms

```
c          callp  p_jsmopen(jsmsrv:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg01:*COMP')
```

* Load the SMSService

* This loads and initializes the service using the values defined in
* the SMSService.properties file.
* This example explicitly turns tracing on.

```
c          eval   jsmcmd = 'SERVICE_LOAD'
c          + ' SERVICE(SMSSERVICE) TRACE(*YES)'
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg02:*COMP')
*
```

* Examples of SET command

* The SET command may be used to override any defaults that are
* defined in the SMSService.properties file.

* Set the encoding value

```
c      eval    jsmcmd = 'SET'
c              + ' ENCODING(ISO8859_1)'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg11:*COMP')
```

* Set the SMTP (email) FROM value.

* The FROM value will be used when sending the email
* to the SMS gateway.

```
c      eval    jsmcmd = 'SET'
c              + ' FROM(' + emailfrom + ')'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg12:*COMP')
```

* SEND the message

```
c      eval    jsmcmd = 'SEND'
c              + ' TO(' + phoneno
c              + ') MSG(' + message + ')'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg20:*COMP')
```

* Unload the SMSService

```
c      eval    jsmcmd = 'SERVICE_UNLOAD'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg98:*COMP')
```

* Close the connection to the JSM server and finish

```
c      callp   p_jsmclose(jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg99:*COMP')
c      eval    *inlr = *on
```

```

c          return

*****
* Procedure to check the result of a Java Service Manager call
*****

p CheckResult  b
d CheckResult  pi
d crjsts          const like(jsmsts)
d crjmsg         const like(jsmmsg)
d crText         s          512a
d crMsg1         c          const('JSM Status : ')
d crMsg2         c          const('JSM Message: ')
d crMsg3         c          const('JSM Service error has +
d                   occurred')
c          if      crjsts <> 'OK'
c          eval   crText = crMsg1 + crjsts
c          callp  SendMessage(crText:*DIAG')
c          eval   crText = crMsg2 + crjmsg
c          callp  SendMessage(crText:*DIAG')
c          callp  SendMessage(crMsg3:*ESCAPE')
c          endif
p CheckResult  e

*****
* Procedure to send a program message
*****

p SendMessage  b

d SendMessage  pi
d smText       512a  VALUE
d smMsgT       10a   VALUE

d smMsgI       s          7a  inz('CPF9897')
d smMsgF       s          20a  inz('QCPFMSG *LIBL  ')
d smDtaL       s          10i 0  inz(%size(smText))
d smStkE       s          10a  inz('*')
d smStkC       s          10i 0  inz(1)
d smMsgK       s          4a
d smErrC       s          10i 0  inz(0)

```

```
c      if      smMsgT = '*ESCAPE'  
c      eval    smMsgI = 'CPF9898'  
c      endif  
  
c      call    'QMHSNDPM'  
c      parm          smMsgI  
c      parm          smMsgF  
c      parm          smText  
c      parm          smDtaL  
c      parm          smMsgT  
c      parm          smStkE  
c      parm          smStkC  
c      parm          smMsgK  
c      parm          smErrC  
  
p      e
```

ZipService Example

This example program will perform the following steps:

1. Performs a series of calls necessary to load the ZipService.
2. Creates a zip file called LIZIPIN.ZIP in directory /LIZIPOUT in the IFS.
3. Zips the contents of directory /LIZIPIN into the zip file and closes the file.
4. Opens the zip file again for reading.
5. Retrieves a list of zip file entries into a multiple occurrence data structure.
6. Unzips the first zip entry into the directory /LIZIPOUT.
7. Closes the zip file, unloads the service and closes the connection to the JSM server.

Note:

- To test this example meaningfully you need to create the two folders /LIZIPIN and /LIZIPOUT in the IFS and then add a small selection of files to the folder /LIZIPIN. You can use any files – these are the files that will be zipped by the example program
- The example program can only successfully retrieve the zip file entries if they number 999 or less, so do not add more files than this to the folder /LIZIPIN
- For maximum compatibility, the example program can only cope with file paths up to 256 characters in length. Do not add files and folders to /LIZIPIN such that any one file path will exceed this limit. You can write your own programs to handle longer paths if necessary.

Refer to the comments and code in the example for more information.

There are three steps required to make this application work:

1. Create the structure XML

The call to the GET ENTRY(*LIST) command of the ZipService passes a multiple occurrence data structure in which it will receive a list of up to 999 zip file entries. For this to work, the LANSA Integrator service needs to know the characteristics of this structure. This is accomplished by supplying an XML file that describes the structure.

For this example, the required XML is supplied below. To install this XML you need to perform the following steps:

- a. Locate the structure folder in the JSM instance folder for your JSM server
- b. Create a file called ZipEntryList.xml
- c. Edit the file with a text editor and paste into it the xml supplied below

Note that the field names used in the structure XML do not need to match the variable names used in the RPG program (and they do not match in this example). It is their order, types and length that are important.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<rdml:structure xmlns:rdml="http://www.lansa.com/2000/XML/Function">
```

```
<rdml:field name="ZIPENT" type="A" length="256" />
```

```
</rdml:structure>
```

2. Register the structure XML with the JSM Server

The example program below refers to the structure XML supplied above with the symbolic name ZIP.ZipEntryList by specifying that name in the SERVICE_STRUCTURE keyword of the GET ENTRY(*LIST) command.

We need to give the JSM Server a link between that symbolic name and the actual name and location of the structure XML file created in step 1. To do this you need to perform the following steps:

- a. Locate the system folder in the JSM instance folder for your JSM server.
- b. Edit the file structure.properties with a text editor and paste into it the entry supplied below (make sure the new entry is on a line by itself).
- c. Save your changes.
- d. Restart or refresh the JSM Server instance (refer to [Java Service Manager Refresh](#)).

```
structure.ZIP.ZipEntryList=structure/ZipEntryList.xml
```

3. Create and run the ILE RPG example program

Recent installations of LANSA Integrator will already contain the source for the example program below. If not, you can copy and paste the source into a source file member.

To create the program, you need to use the CRTRPGMOD and CRTPGM commands. Make sure that you use the parameter values specified in the source member.

* ZIP: example in RPG ILE of using the LANSA Integrator
* ZipService

*

* Note: This is an example program containing only
* rudimentary exception handling

*

* To create this program you must execute the following commands,
* supplying the indicated parameter values and any others that are
* necessary in your installation:

*

* CRTRPGMOD MODULE(<modlib>/ZIP)
* SRCFILE(<srclib>/<srcfil>)

*

* CRTPGM PGM(<pgmlib>/ZIP)
* MODULE(<modlib>/ZIP)
* BNDSRVPGM(<jsmpgmlib>/DCXS882X)
* ACTGRP(*CALLER)

* IFS folders used by this program
* - to try this program you need to create these folders in your IFS
* and add one or more files to the /LIZIPIN folder

```
d flrzipin      c          const('/LIZIPIN')
d flrzipout    c          const('/LIZIOUT')
d zipfilepath  s          255a
```

* Declare variables for the JSM calls

```
d jsmsrv       s          50a inz(*blanks)
d jsmsts       s          20a inz(*blanks)
d jsmsg        s          255a inz(*blanks)
d jscommand   s          255a inz(*blanks)
d bytelength   s          10i 0 inz(*zero)
```

* Declare structure to send or receive zip file entries
* - the structure must contain the following fields as defined
* by the GET command of the ZipService

- * o Zip entry (path and/or file names)
- * NB: This MUST match the structure xml provided to the JSM Server!

```

d ziplist      ds          occurs(zipocur)
d zipentry    256a
d zipocur     c          const(999)
d zipsize     c          const(%size(ziplist))
d zipcount    s          9p 0 inz(0)

```

* Completion messages

```

d CompMsg10   c          'JSMOPEN call completed.'
d CompMsg20   c          ' SERVICE_LOAD call completed.'
d CompMsg30   c          ' CREATE call completed.'
d CompMsg40   c          ' OPEN call completed.'
d CompMsg50   c          ' ADD call completed.'
d CompMsg60   c          ' GET ENTRY(*LIST) call completed.'
d CompMsg65   c          ' GET ENTRY(*READ) call completed.'
d CompMsg70   c          ' CLOSE call completed.'
d CompMsg80   c          ' SERVICE_UNLOAD call completed.'
d CompMsg99   c          'JSMCLOSE call completed.'

```

* Procedure prototypes

```

d CheckResult pr
d crjsts      const like(jsmsts)
d crjmsg      const like(jsmmsg)

d SendMessage pr
d smText      512a VALUE
d smType      10a VALUE

```

* Prototypes for the JSM calls

/COPY QRPGLSRC,JSM_PROC.H

- * Open a connection to the default JSM server
- * - because the server parameter is blank, details of the default

- * JSM server are obtained from the data area JSMCTLDTA on IBM i
- * or from the file jsmctldta.txt on other supported platforms)

```

C          callp  p_jsmopen(jsmsrv:jsmsts:jsmmsg)
C          callp  CheckResult(jsmsts:jsmmsg)
C          callp  SendMessage(CompMsg10:*COMP')

```

* Load the ZipService

- * - this example explicitly turns tracing on, overriding the
- * settings in the manager.properties file

```

C          eval   jsmcmd = 'SERVICE_LOAD'
C          + ' SERVICE(ZIPSERVICE) TRACE(*YES)'
C          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
C          callp  CheckResult(jsmsts:jsmmsg)
C          callp  SendMessage(CompMsg20:*COMP')

```

* Create the zip file: /LIZIPOUT/LIZIPIN.ZIP

- * - we place it in the /LIZIPOUT folder
- * - we will zip the contents of /LIZIPIN into it

```

C          eval   zipfilepath = flrzipout + flrzipin + '.ZIP'
C          eval   jsmcmd = 'CREATE'
C          + ' FILE(' + %trim(zipfilepath) + ')'
C          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
C          callp  CheckResult(jsmsts:jsmmsg)
C          callp  SendMessage(CompMsg30:*COMP')

```

* Add the contents of the folder /LIZIPIN to the zip file

- * - by specifying BASE(*CURRENT) we request that path information stored
- * in the zip entries is only for descendant folders of /LIZIPIN

```

C          eval   jsmcmd = 'ADD'
C          + ' PATH(' + flrzipin + ') BASE(*CURRENT)'
C          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
C          callp  CheckResult(jsmsts:jsmmsg)
C          callp  SendMessage(CompMsg50:*COMP')

```

* Close the current archive

```

c          eval    jsmcmd = 'CLOSE'
c          callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg70:*COMP')

```

* Reopen the zip file: /LIZIPOUT/LIZIPIN.ZIP

* - we will retrieve a list of its contents

* - we will unzip the first file into /LIZIPOUT

```

c          eval    jsmcmd = 'OPEN'
c                    + ' FILE(' + %trim(zipfilepath) + ')'
c          callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg40:*COMP')

```

* Load a list of zip entries from the current archive

* - although this passes the multiple occurrence data structure

* (ziplist) in which to receive the list, the data is not actually

* received into the structure until the JSMBYTERECV call below

* - the structure is described to the ZipService by the

* structure XML identified by the SERVICE_STRUCTURE keyword - there

* must be a matching entry in the structure.properties file and a

* corresponding structure XML file, usually in the JSMInstance\Structure

* folder

* NOTE: this call uses the JSMCMDX api in order to be able to send and/or

* receive variable data (in this case the list)

```

c          eval    jsmcmd = 'GET ENTRY(*LIST)'
c                    + ' SERVICE_STRUCTURE(ZIP.ZipEntryList)'
c                    + ' OCCURS(' + %char(zipocur) + ')'
c                    + ' SIZE(' + %char(zipsize) + ')'

```

```

c          eval    %occur(ziplist) = 1
c          eval    bytelength = zipocur * zipsize

```

```

c          callp   p_jsmcmdx(jsmcmd:ziplist:bytelength:
c                    jsmsts:jsmmsg)

```

```

c          callp  CheckResult(jsmsts:jmmmsg)
c          callp  SendMessage(CompMsg60:*COMP')

* Get the length of the received data structure
* - only continue to attempt to receive it if the length is valid

c          callp  p_jsmbytelngth(bytelength)
c          if    (bytelength > 0)
c              and (bytelength <= zipocur * zipsize)

* ... receive the zip entry list into
* our multiple occurrence data structure
* - don't forget to set the DS occurrence to 1 before the call

c          callp  p_jsmbyterecev(ziplist)

* ... calculate the number of entries

c          eval  zipcount = bytelength / zipsize

* ... unzip the first entry only into /LIZIPOUT

c          eval  jsmcmd = 'GET ENTRY(*READ)'
c              + ' FILE(' + %trim(zipentry) + ')'
c              + ' TO(' + flrzipout + ')'
c          callp  p_jsmcmd(jsmcmd:jsmsts:jmmmsg)
c          callp  CheckResult(jsmsts:jmmmsg)
c          callp  SendMessage(CompMsg65:*COMP')

c          endif

* Close the current archive

c          eval  jsmcmd = 'CLOSE'
c          callp  p_jsmcmd(jsmcmd:jsmsts:jmmmsg)
c          callp  CheckResult(jsmsts:jmmmsg)
c          callp  SendMessage(CompMsg70:*COMP')

* Unload the ZipService

```

```

c      eval    jsmcmd = 'SERVICE_UNLOAD'
c      callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg80:*COMP')

```

* Close the connection to the JSM server and finish

```

c      callp   p_jsmclose(jsmsts:jsmmsg)
c      callp   CheckResult(jsmsts:jsmmsg)
c      callp   SendMessage(CompMsg99:*COMP')
c      eval    *inlr = *on
c      return

```

* Procedure to check the result of a Java Service Manager call

```

p CheckResult  b
d CheckResult  pi
d crjsts              const like(jsmsts)
d crjmsg            const like(jsmmsg)
d crText           s      512a
d crMsg1           c      const('JSM Status : ')
d crMsg2           c      const('JSM Message: ')
d crMsg3           c      const('JSM Service error has +
d                   occurred')
c      if      crjsts <> 'OK'
c      eval    crText = crMsg1 + crjsts
c      callp   SendMessage(crText:*DIAG')
c      eval    crText = crMsg2 + crjmsg
c      callp   SendMessage(crText:*DIAG')
c      callp   SendMessage(crMsg3:*ESCAPE')
c      endif
p CheckResult  e

```

* Procedure to send a program message

```

p SendMessage  b

```

```

d SendMessage pi
d smText      512a VALUE
d smMsgT      10a  VALUE

d smMsgI      s      7a  inz('CPF9897')
d smMsgF      s      20a  inz('QCPFMSG *LIBL  ')
d smDtaL      s      10i 0  inz(%size(smText))
d smStkE      s      10a  inz('*')
d smStkC      s      10i 0  inz(1)
d smMsgK      s      4a
d smErrC      s      10i 0  inz(0)

c      if      smMsgT = '*ESCAPE'
c      eval    smMsgI = 'CPF9898'
c      endif

c      call    'QMHSNDPM'
c      parm    smMsgI
c      parm    smMsgF
c      parm    smText
c      parm    smDtaL
c      parm    smMsgT
c      parm    smStkE
c      parm    smStkC
c      parm    smMsgK
c      parm    smErrC

p      e

```

PDFSpoolFileService Example

This example program will perform the following steps:

1. Executes an OVRPRTF command and a WRKJOB command to create a spooled file on output queue QPRINT in QGPL with user data PDFSPLF.
2. Performs a series of calls necessary to load the PDFSpoolFileService and connect to the IBM i host machine.
3. Retrieves a list of spool files from output queue QPRINT in QGPL with user data PDFSPLF.
4. Converts the first spool file in the list to the PDF file PDFSPLF.PDF in the JSM instance root folder (this may not be the spool file created by the current run if the program has been run before without cleaning up previous spool files).
5. Disconnects, unloads the service and closes the connection to the JSM server.

Refer to the comments and code in the example for more information.

There are three steps required to make this application work:

1. Create the structure XML

The call to the LIST command of the PDFSpoolFileService passes a multiple occurrence data structure in which it will receive a list of up to 50 spool files on the specified output queue. For this to work, the LANSA Integrator service needs to know the characteristics of this structure. This is accomplished by supplying an XML file that describes the structure.

For this example, the required XML is supplied below. Recent installations of LANSA Integrator will already contain this structure XML. If not you need to perform the following steps:

- a. Locate the structure folder in the JSM instance folder for your JSM server.
- b. Create a file called PDFSpoolFileList.xml.
- c. Edit the file with a text editor and paste into it the xml supplied below.

Note that the field names used in the structure XML do not need to match the variable names used in the RPG program (and they do not match in this example). It is their order, types and length that are important.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<rdml:structure xmlns:rdml="http://www.lansa.com/2000/XML/Function">

  <rdml:field name="SPFNAM" type="A" length="10" />
  <rdml:field name="SPFNUM" type="A" length="6" />
  <rdml:field name="JOBNAM" type="A" length="10" />
  <rdml:field name="JOBUSR" type="A" length="10" />
  <rdml:field name="JOBNUM" type="A" length="6" />
  <rdml:field name="SPFSTS" type="A" length="10" />

</rdml:structure>

```

3. Register the structure XML with the JSM Server

The example program below refers to the structure XML supplied above with the symbolic name PDFSPLF.SpoolFileList by specifying that name in the SERVICE_STRUCTURE keyword of the LIST command.

We need to give the JSM Server a link between that symbolic name and the actual name and location of the structure XML file created in step 1. Recent installations of LANSA Integrator will already contain this link. Otherwise, you need to perform the following steps:

- a. Locate the system folder in the JSM instance folder for your JSM server.
- b. Edit the file structure.properties with a text editor and paste into it the entry supplied below (make sure the new entry is on a line by itself).
- c. Save your changes.
- d. Restart or refresh the JSM Server instance (refer to [Java Service Manager Refresh](#)).

```
structure.PDFSPLF.SpoolFileList=structure/PDFSpoolFileList.xml
```

3. Create and run the ILE RPG example program

Recent installations of LANSA Integrator will already contain the source for the example program below. If not, you can copy and paste the source into a source file member.

To create the program, you need to use the CRTRPGMOD and CRTPGM commands. Make sure that you use the parameter values specified in the source member.

```

*****
* PDFSPLF: example in RPG ILE of using the LANSA Integrator
*   PDFSpoolFileService

```

*
 * Note: This is an example program containing only
 * rudimentary exception handling
 *
 * To create this program you must execute the following commands,
 * supplying the indicated parameter values and any others that are
 * necessary in your installation:

```
* CRTRPGMOD MODULE(<modlib>/PDFSPLF)
* SRCFILE(<srclib>/<srcfil>)
*
* CRTPGM PGM(<pgmlib>/PDFSPLF)
* MODULE(<modlib>/PDFSPLF)
* BNDSRVPGM(<jsmpgmlib>/DCXS882X)
* ACTGRP(*CALLER)
```

* Commands to create the spooled file used by this program

```
d ovrprt      c          const('OVRPRTF FILE(QPDSPJOB) +
d              OUTQ(QGPL/QPRINT) HOLD(*YES) +
d              USRDTA(PDFSPLF)')
d wrkjob      c          const('WRKJOB OUTPUT(*PRINT)')
d ovrprtlen   c          const(%len(ovrprt))
d wrkjoblen   c          const(%len(wrkjob))
```

* Declare variables for the JSM calls

```
d jsmsrv      s          50a inz(*blanks)
d jsmsts      s          20a inz(*blanks)
d jsmsg       s          255a inz(*blanks)
d jsmcmd      s          255a inz(*blanks)
d bytelength  s          10i 0 inz(*zero)
```

* Declare structure to receive spool file list

* - the structure must contain the following fields as defined
 * by the LIST command of the PDFSpoolFileService
 * o Spool file name
 * o Spool file number

- * o Job name
- * o Job user
- * o Job number
- * o Spool file status
- * NB: This MUST match the structure xml provided to the JSM Server!

```

d spoollist      ds          occurs(spoolocur)
d spoolfnam      10a
d spoolfnum      6a
d spooljnam      10a
d spooljusr      10a
d spooljnum      6a
d spoolfst       10a

d spoolocur      c          const(50)
d spoolsize      c          const(%size(spoollist))
d spoolcount     s          9p 0 inz(0)

```

* Completion messages

```

d CompMsg10      c          'JSMOPEN call completed.'
d CompMsg20      c          ' SERVICE_LOAD call completed.'
d CompMsg30      c          ' CONNECT call completed.'
d CompMsg40      c          ' LIST call completed.'
d CompMsg50      c          ' CREATE call completed.'
d CompMsg60      c          ' DISCONNECT call completed.'
d CompMsg70      c          ' SERVICE_UNLOAD call completed.'
d CompMsg99      c          'JSMCLOSE call completed.'

```

* Procedure prototypes

```

d CheckResult    pr
d crjsts          const like(jsmsts)
d crjmsg          const like(jsmmsg)

d SendMessage     pr
d smText          512a VALUE
d smType          10a VALUE

```

* Prototypes for the JSM calls

```
/COPY QRPGLSRC,JSM_PROC.H
```

* Create a spool file that we will convert to PDF

```
c      call  'QCMDEXC'  
c      parm  ovrprt      command      255  
c      parm  ovrprtlen   commandlen  15 5  
  
c      call  'QCMDEXC'  
c      parm  wrkjob      command      255  
c      parm  wrkjoblen   commandlen  15 5
```

* Open a connection to the default JSM server

* - because the server parameter is blank, details of the default

* JSM server are obtained from the data area JSMCTLDTA on IBM i

* or from the file jsmctldta.txt on other supported platforms)

```
c      callp  p_jsmopen(jsmsrv:jsmsts:jsmmsg)  
c      callp  CheckResult(jsmsts:jsmmsg)  
c      callp  SendMessage(CompMsg10:*COMP')
```

* Load the PDFSpoolFileService

* - this example explicitly turns tracing on, overriding the

* settings in the manager.properties file

```
c      eval   jsmcmd = 'SERVICE_LOAD'  
c      + ' SERVICE(PDFSPOOLFILESERVICE) TRACE(*YES)  
c      callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)  
c      callp  CheckResult(jsmsts:jsmmsg)  
c      callp  SendMessage(CompMsg20:*COMP')
```

* Connect to the IBM i host machine containing the required spool files

* - because no HOST parameter is specified, the service will establish

* the connection to the same IBM i that the JSM Server is running on

```
c      eval   jsmcmd = 'CONNECT'  
c      callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
```

```
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg30:*COMP')
```

* Load a list of spool files on the QGPL/QPRINT output queue
* - although this passes the multiple occurrence data structure
* (spoollist) in which to receive the list, the data is not actually
* received into the structure until the JSMBYTERECV call below
* - the structure is described to the PDFSpoolFileService by the
* structure XML identified by the SERVICE_STRUCTURE keyword - there
* must be a matching entry in the structure.properties file and a
* corresponding structure XML file, usually in the JSMInstance\Structure
* folder

* NOTE: this call uses the JSMCMDX api in order to be able to send and/or
* receive variable data (in this case the list)

```
c          eval    jsmcmd = 'LIST'
c          + ' LIBRARY(QGPL) QUEUE(QPRINT)'
c          + ' USERDATA(PDFSPLF)'
c          + ' SERVICE_STRUCTURE('
c          + ' PDFSPLF.SpoolFileList)'
c          + ' OCCURS(' + %char(spoolocur) + ')'
c          + ' SIZE(' + %char(spoolsz) + ')'
```

```
c          eval    %occur(spoollist) = 1
c          eval    bytelength = spoolocur * spoolsz
```

```
c          callp   p_jsmcmdx(jsmcmd:spoollist:bytelength:
c                  jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg40:*COMP')
```

* Get the length of the received data structure
* - only continue to attempt to receive it if the length is valid

```
c          callp   p_jsmbytelngth(bytelength)
c          if      (bytelength > 0)
c          and (bytelength <= spoolocur * spoolsz)
```

- * ... receive the spool file list into
- * our multiple occurrence data structure
- * - don't forget to set the DS occurrence to 1 before the call

```
c          callp    p_jsmbyterecv(spoollist)
```

- * ... calculate the number of entries
- * - for illustration only, not used in this example program

```
c          eval     spoolcount = bytelength / spoolsize
```

- * ... create a PDF file in the root of the IFS for the first
- * spool file (if any)

```
c          eval     jsmcmd = 'CREATE'
c          + ' DOCUMENT(PDFSPLF.PDF)'
c          + ' NAME(' + %trim(spoolfnam) + ')'
c          + ' JOBNAME(' + %trim(spooljnam) + ')'
c          + ' JOBUSER(' + %trim(spooljusr) + ')'
c          + ' JOBNUMBER(' + %trim(spooljnum) + ')'
c          + ' NUMBER(' + %trim(spoolfnum) + ')'
c          + ' FONTSIZE(8) LEADING(8.5)'
c          + ' ORIENTATION(*LANDSCAPE)'
c          callp    p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp    CheckResult(jsmsts:jsmmsg)
c          callp    SendMessage(CompMsg50:*COMP')

c          endif
```

- * Close the current IBM i host machine connection

```
c          eval     jsmcmd = 'DISCONNECT'
c          callp    p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp    CheckResult(jsmsts:jsmmsg)
c          callp    SendMessage(CompMsg60:*COMP')
```

- * Unload the PDFSpoolFileService

```
c          eval     jsmcmd = 'SERVICE_UNLOAD'
```

```
c      callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp  CheckResult(jsmsts:jsmmsg)
c      callp  SendMessage(CompMsg70:*COMP')
```

* Close the connection to the JSM server and finish

```
c      callp  p_jsmclose(jsmsts:jsmmsg)
c      callp  CheckResult(jsmsts:jsmmsg)
c      callp  SendMessage(CompMsg99:*COMP')
c      seton          LR
c      return
```

* Procedure to check the result of a Java Service Manager call

```
p CheckResult  b
d CheckResult  pi
d crjsts          const like(jsmsts)
d crjmsg         const like(jsmmsg)
d crText        s      512a
d crMsg1        c      const('JSM Status : ')
d crMsg2        c      const('JSM Message: ')
d crMsg3        c      const('JSM Service error has +
d                  occurred')
c      if      crjsts <> 'OK'
c      eval    crText = crMsg1 + crjsts
c      callp   SendMessage(crText:*DIAG')
c      eval    crText = crMsg2 + crjmsg
c      callp   SendMessage(crText:*DIAG')
c      callp   SendMessage(crMsg3:*ESCAPE')
c      endif
p CheckResult  e
```

* Procedure to send a program message

```
p SendMessage  b
d SendMessage  pi
```

```

d smText          512a VALUE
d smMsgT          10a VALUE

d smMsgI          s      7a inz('CPF9897')
d smMsgF          s      20a inz('QCPFMSG *LIBL  ')
d smDtaL          s      10i 0 inz(%size(smText))
d smStkE          s      10a inz('*')
d smStkC          s      10i 0 inz(1)
d smMsgK          s      4a
d smErrC          s      10i 0 inz(0)

c      if      smMsgT = '*ESCAPE'
c      eval    smMsgI = 'CPF9898'
c      endif

c      call    'QMHSNDPM'
c      parm    smMsgI
c      parm    smMsgF
c      parm    smText
c      parm    smDtaL
c      parm    smMsgT
c      parm    smStkE
c      parm    smStkC
c      parm    smMsgK
c      parm    smErrC

p      e

```

SVFileService Example

This example program performs the following steps:

1. Performs a series of calls necessary to load the SVFileService
2. Writes a CSV file containing order line data to file SVFILE.CSV in the JSM instance folder – in this simple example, the data comes from compile-time array data coded in the RPG program, but it could equally well have come from a database or have been received in various formats through another LANSA Integrator service call
3. Unloads the service and closes the connection to the JSM server.

Refer to the comments and code in the example for more information.

There are four steps required to make this application work:

1. Create the structure XML

The call to the WRITE command of the SVFileService passes a multiple occurrence data structure containing the order line items that will be written to the CSV file. For this to work, the LANSA Integrator service needs to know the characteristics of this structure. This is accomplished by supplying an XML file that describes the structure.

For this example, the required XML is supplied below. To install this XML you need to perform the following steps:

- a. Locate the structure folder in the JSM instance folder for your JSM server.
- b. Create a file called SVOrderLine.xml.
- c. Edit the file with a text editor and paste into it the xml supplied below.

Note that the field names used in the structure XML do not need to match the variable names used in the RPG program (although they do match in this example). It is their order, types and length that are important – not their names.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<rdml:structure xmlns:rdml="http://www.lansa.com/2000/XML/Function">
```

```
<rdml:field name="LINENUM" type="S" length="7" />
```

```
<rdml:field name="PARTNUM" type="A" length="7" />
```

```
<rdml:field name="PARTDSC" type="A" length="30" />
```

```
<rdml:field name="PARTAMT" type="S" length="9" decimal="2" />
```

```
<rdml:field name="PARTQTY" type="S" length="7" />
```

```
</rdml:structure>
```

2. Register the structure XML with the JSM Server

The example program refers to the structure XML supplied above with the symbolic name SV.SVOrderLine by specifying that name in the SERVICE_STRUCTURE keyword of the WRITE command.

We need to give the JSM Server a link between that symbolic name and the actual name and location of the structure XML file created in step 1. To do this you need to perform the following steps:

- a. Locate the system folder in the JSM instance folder for your JSM server.
- b. Edit the file structure.properties with a text editor and paste into it the entry supplied below (make sure the new entry is on a line by itself).
- c. Save your changes.
- d. Restart or refresh the JSM Server instance (refer to [Java Service Manager Refresh](#)).

```
structure.SV.SVOrderLine=structure/SVOrderLine.xml
```

3. Define the header for the resulting CSV

The contents of the SVFILE.CSV file that is created by this sample should look similar to this:

```
LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY  
1,123,Gasket Paper,9.95,10  
2,456,Gasket polymer glue,13.27,5
```

You will note that the first record contains column names (or headings). Many programs that process CSV data will or can recognize and make use of these column names.

The SVFileService creates these column headings optionally. If you wish to include them, then you need to specify the SVHEAD parameter of the WRITE command.

This example, specifies the symbolic name ORDER for the SVHEAD parameter. This symbolic name identifies a set of column headings that are specified in the service properties file for the SVFileService. Therefore you need to perform the following steps to add the definition to the service properties file (if it is not already there):

- a. Locate the properties folder in the JSM instance folder for your JSM server
- b. Edit the file SVFileService.properties with a text editor and paste into it the entry supplied below, if it is not already there (make sure that the new entry is on a line by itself)
- c. Save your changes
- d. Restart or refresh the JSM Server instance (refer to [Java Service Manager Refresh](#)).

Note that the column headings used do not need to match the variable names used in the RPG program or in the structure XML (although they do match in this example).

```
sv.head.order=LINENUM,PARTNUM,PARTDSC,PARTAMT,PARTQTY
```

4. Create and run the ILE RPG example program

Copy and paste the source provided below into a source file member.

To create the program, you need to use the CRTRPGMOD and CRTPGM commands. Make sure that you use the parameter values specified in the source member.

```
*****
* SVFILE: example in RPG ILE of using the LANSA Integrator
*   SVFileService to write a comma-separated file
*
* Note: This is an example program containing only
*   rudimentary exception handling
*
* To create this program you must execute the following commands,
* supplying the indicated parameter values and any others that are
* necessary in your installation:
*
* CRTRPGMOD MODULE(<modlib>/SVFILE)
*   SRCFILE(<srclib>/<srcfil>)
*
* CRTPGM PGM(<pgmlib>/SVFILE)
*   MODULE(<modlib>/SVFILE)
*   BNDSRVPGM(<jsmpgmlib>/DCXS882X)
*   ACTGRP(*CALLER)
*****
```

- * Path of the CSV file created by this program
- * - because no folder path is specified, the file will be created
- * in the JSM instance folder by default

```
d svfilepath    c                const('SVFILE.CSV')
```

- * Declare variables for the JSM calls

```
d jsmsrv       s                50a  inz(*blanks)
d jsmsts       s                20a  inz(*blanks)
d jsmmsg       s                255a  inz(*blanks)
d jsmcmd       s                255a  inz(*blanks)
d bytelength   s                10i 0  inz(*zero)
```

- * Declare structure to send order line data to be written to the
- * comma-separated file:
- * - in this simple example, the data comes from the compile-time
- * array data, but it could equally well have come from a database
- * or received through another LANSAs Integrator service call
- * NB: This MUST match the structure xml provided to the JSM Server!

```
d svlist       ds                occurs(svocur) based(svlistptr)
d linenum      7s 0
d partnum      7a
d partdsc      30a
d partamt      9s 2
d partqty      7s 0
```

```
d svocur       c                const(2)
d svsize       c                const(%size(svlist))
```

- * Declare the compile-time array that provides the data for
- * this simple example

```
d svdata       s                60a  dim(svocur) perrcd(1) ctdata
```

- * Completion messages

```
d CompMsg01    c          'JSMOPEN call completed.'
d CompMsg02    c          ' SERVICE_LOAD call completed.'
d CompMsg10    c          ' WRITE call completed.'
d CompMsg98    c          ' SERVICE_UNLOAD call completed.'
d CompMsg99    c          'JSMCLOSE call completed.'
```

* Procedure prototypes

```
d CheckResult  pr
d crjsts              const like(jsmsts)
d crjmsg           const like(jsmmsg)
```

```
d SendMessage   pr
d smText        512a  VALUE
d smType        10a   VALUE
```

* Prototypes for the JSM calls

```
/COPY QRPGLSRC,JSM_PROC.H
```

* Open a connection to the default JSM server
* - because the server parameter is blank, details of the default
* JSM server are obtained from the data area JSMCLTDTA on IBM i
* or from the file jsmcltdta.txt on other supported platforms)

```
c          callp  p_jsmopen(jsmsrv:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg01:*COMP')
```

* Load the SVFileService
* - this example explicitly turns tracing on, overriding the
* settings in the manager.properties file

```
c          eval   jsmcmd = 'SERVICE_LOAD'
c          + ' SERVICE(SVFILESERVICE) TRACE(*YES)'
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg02:*COMP')
```

- * Populate the list to be written to the CSV file
- * - in this simple example, the data comes from the compile-time
- * array data, but it could equally well have come from a database
- * or received through another LANSAs Integrator service call

```
c          eval    svlistptr = %addr(svdata)
```

- * Write a comma-separated list of items from the compile-time array
- * - the SVHEAD parameter identifies headings (defined in the SVFILESEI
- * service properties file) that will be written to the CSV file
- * - this passes the multiple occurrence data structure
- * (svlist) containing the items
- * - the structure is described to the SVFileService by the
- * structure XML identified by the SERVICE_STRUCTURE keyword - th
- * must be a matching entry in the structure.properties file and a
- * corresponding structure XML file, usually in the <instance>\Structure
- * folder

- * NOTE: this call uses the JSMCMDX api in order to be able to send
- * variable data (in this case the structure/list)

```
c          eval    jsmcmd = 'WRITE'
c          + ' FILE(' + %trim(svfilepath) + ')'
c          + ' SVHEAD(ORDER)'
c          + ' SERVICE_STRUCTURE(SV.SVOrderLine)'
c          + ' COUNT(' + %char(svocur) + ')'
c          + ' OCCURS(' + %char(svocur) + ')'
c          + ' SIZE(' + %char(svsize) + ')'
```

```
c          eval    bytelength = svocur * svsize
```

```
c          callp   p_jsmcmdx(jsmcmd:svlist:bytelength:
c                  jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg10:*COMP')
```

- * Unload the SVFileService

```
c          eval    jsmcmd = 'SERVICE_UNLOAD'
```

```

c      callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c      callp  CheckResult(jsmsts:jsmmsg)
c      callp  SendMessage(CompMsg98:*COMP')

```

* Close the connection to the JSM server and finish

```

c      callp  p_jsmclose(jsmsts:jsmmsg)
c      callp  CheckResult(jsmsts:jsmmsg)
c      callp  SendMessage(CompMsg99:*COMP')
c      eval   *inlr = *on
c      return

```

* Procedure to check the result of a Java Service Manager call

```

p CheckResult  b
d CheckResult  pi
d crjsts              const like(jsmsts)
d crjmsg           const like(jsmmsg)
d crText          s      512a
d crMsg1          c      const('JSM Status : ')
d crMsg2          c      const('JSM Message: ')
d crMsg3          c      const('JSM Service error has +
d                  occurred')
c      if      crjsts <> 'OK'
c      eval    crText = crMsg1 + crjsts
c      callp   SendMessage(crText:*DIAG')
c      eval    crText = crMsg2 + crjmsg
c      callp   SendMessage(crText:*DIAG')
c      callp   SendMessage(crMsg3:*ESCAPE')
c      endif
p CheckResult  e

```

* Procedure to send a program message

```

p SendMessage  b

d SendMessage  pi

```

```

d smText          512a VALUE
d smMsgT          10a VALUE

d smMsgI          s      7a inz('CPF9897')
d smMsgF          s      20a inz('QCPFMSG *LIBL  ')
d smDtaL          s      10i 0 inz(%size(smText))
d smStkE          s      10a inz('*')
d smStkC          s      10i 0 inz(1)
d smMsgK          s      4a
d smErrC          s      10i 0 inz(0)

```

```

c      if      smMsgT = '*ESCAPE'
c      eval    smMsgI = 'CPF9898'
c      endif

```

```

c      call    'QMHSNDPM'
c      parm    smMsgI
c      parm    smMsgF
c      parm    smText
c      parm    smDtaL
c      parm    smMsgT
c      parm    smStkE
c      parm    smStkC
c      parm    smMsgK
c      parm    smErrC

```

```

p      e
**CTDATA svdata
0000001123 Gasket Paper          0000009950000010
0000002456 Gasket polymer glue   0000013270000005

```

ExcelReadService Example

This example program will perform the following steps:

1. It performs a series of calls necessary to load the ExcelReadService.
2. It opens, writes and closes an Excel workbook file containing order line data to file XLREAD.XLS in the JSM instance folder – in this simple example, the data comes from compile-time array data coded in the RPG program, but it could equally well have come from a database or have been received in various formats through another LANSA Integrator service call.
3. It unloads the service and closes the connection to the JSM server.

Refer to the comments and code in the example for more information.

There are three steps required to make this application work:

1. Create the structure XML

The calls to the WRITE command of the ExcelReadService pass a multiple occurrence data structure containing the order line items that will be written to the workbook. For this to work, the LANSA Integrator service needs to know the characteristics of the structure. This is accomplished by supplying an XML file that describes the structure.

For this example, two structures are used – one is used to write column headings one cell at a time while the second is used to write the order line data. The required XML is supplied below. To install this XML you need to perform the following steps:

- a. Locate the structure folder in the JSM instance folder for your JSM server
- b. Create files called XLCell.xml and XLOrderLine.xml
- c. Edit the XLCell.xml file with a text editor and paste into it the xml supplied below

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<rdml:structure xmlns:rdml="http://www.lansa.com/2000/XML/Function">
```

```
  <rdml:field name="XLCELL" type="A" length="256" />
```

```
</rdml:structure>
```

- d. Edit the XLOrderLine.xml file with a text editor and paste into it the xml

supplied below

```
<?xml version="1.0" encoding="UTF-8"?>  
  
<rdml:structure xmlns:rdml="http://www.lansa.com/2000/XML/Function">  
  
  <rdml:field name="LINENUM" type="S" length="7" />  
  <rdml:field name="PARTNUM" type="A" length="7" />  
  <rdml:field name="PARTDSC" type="A" length="30" />  
  <rdml:field name="PARTAMT" type="S" length="9" decimal="2" />  
  <rdml:field name="PARTQTY" type="S" length="7" />  
  
</rdml:structure>
```

Note that the field names used in the structure XML do not need to match the variable names used in the RPG program. It is their order, types and length that are important – not their names.

2. Register the structure XML with the JSM Server

The example program refers to the structure XML supplied above with the symbolic names XL.XLCell and XL.XLOrderLine by specifying those names in the SERVICE_STRUCTURE keyword of the WRITE command.

We need to give the JSM Server a link between those symbolic names and the actual names and locations of the structure XML files created in step 1. To do this you need to perform the following steps:

- a. Locate the system folder in the JSM instance folder for your JSM server
- b. Edit the file structure.properties with a text editor and paste into it the entries supplied below (make sure the new entries are each on separate lines by themselves)
- c. Save your changes
- d. Restart or refresh the JSM Server instance (refer to [Java Service Manager Refresh](#)).

```
structure.XL.XLCell=structure/XLCell.xml  
structure.XL.XLOrderLine=structure/XLOrderLine.xml
```

3. Create and run the ILE RPG example program

Copy and paste the source provided below into a source file member.

To create the program, you need to use the CRTRPGMOD and CRTPGM

commands. Make sure that you use the parameter values specified in the source member.

```
*****
* EXCELREAD: example in RPG ILE of using the LANSA Integrator
*   ExcelReadService to write an Excel workbook.
*
* Note:  This is an example program containing only
*   rudimentary exception handling
*
* To create this program you must execute the following commands,
* supplying the indicated parameter values and any others that are
* necessary in your installation:
*
* CRTRPGMOD MODULE(<modlib>/EXCELREAD)
*   SRCFILE(<srclib>/<srcfil>)
*
* CRTPGM  PGM(<pgmlib>/EXCELREAD)
*   MODULE(<modlib>/EXCELREAD)
*   BNDSRVPGM(<jsmplib>/DCXS882X)
*   ACTGRP(*CALLER)
*****
```

```
* Path of the Excel workbook created by this program
* - because no folder path is specified, the file will be created
*   in the JSM instance folder by default
```

```
d xlfilepath    c                const('XLREAD.XLS')
```

```
* Declare variables for the JSM calls
```

```
d jsmsrv      s          50a  inz(*blanks)
d jsmsts      s          20a  inz(*blanks)
d jsmsg       s          255a  inz(*blanks)
d jscommand   s          255a  inz(*blanks)
d bytelength  s           10i  0  inz(*zero)
```

```
* Declare structure to send order line data to be written to the
* Excel workbook:
```

- * - in this simple example, the data comes from the compile-time
- * array data, but it could equally well have come from a database
- * or received through another LANSAs Integrator service call
- * NB: This MUST match the structure xml provided to the JSM Server!

```
d xllist      ds          occurs(xlocur) based(xllistptr)
d linenum    7s 0
d partnum    7a
d partdsc    30a
d partamt    9s 2
d partqty    7s 0
```

```
d xlocur     c          const(2)
d xlsize     c          const(%size(xllist))
```

- * Declare the compile-time array that provides the data for
- * this simple example

```
d xldata     s          60a dim(xlocur) perrcd(1) ctdata
```

- * Completion messages

```
d CompMsg01  c          'JSMOPEN call completed.'
d CompMsg02  c          ' SERVICE_LOAD call completed.'
d CompMsg10  c          ' OPEN call completed.'
d CompMsg20  c          ' WRITE call completed.'
d CompMsg30  c          ' CLOSE call completed.'
d CompMsg98  c          ' SERVICE_UNLOAD call completed.'
d CompMsg99  c          'JSMCLOSE call completed.'
```

- * Procedure prototypes

```
d WriteCell  pr
d wcR1C1     10a value
d wcCell     256a value
```

```
d CheckResult pr
d crjsts     const like(jsmsts)
d crjmsg     const like(jsmmsg)
```

```
d SendMessage    pr
d smText         512a VALUE
d smType         10a  VALUE
```

* Prototypes for the JSM calls

```
/COPY QRPGLSRC,JSM_PROC.H
```

* Open a connection to the default JSM server
* - because the server parameter is blank, details of the default
* JSM server are obtained from the data area JSMCLDTA on IBM i
* or from the file jsmcltda.txt on other supported platforms)

```
c          callp  p_jsmopen(jsmsrv:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg01:*COMP')
```

* Load the ExcelReadService
* - this example explicitly turns tracing on, overriding the
* settings in the manager.properties file

```
c          eval   jsmcmd = 'SERVICE_LOAD'
c          + ' SERVICE(EXCELREADSERVICE) TRACE(*YES)
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg02:*COMP')
```

* Open the Excel workbook for writing

```
c          eval   jsmcmd = 'OPEN'
c          + ' FILE(' + %trim(xlfilepath) + ')'
c          + ' MODE(*WRITE) REPLACE(*YES)'
c          callp  p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg10:*COMP')
```

* Write some column headings

```

c          callp  WriteCell('3,3':'Line Number')
c          callp  WriteCell('3,4':'Part Number')
c          callp  WriteCell('3,5':'Part Description')
c          callp  WriteCell('3,6':'Unit Amount')
c          callp  WriteCell('3,7':'Unit Quantity')

```

- * Populate the list to be written to the Excel workbook
- * - in this simple example, the data comes from the compile-time
- * array data, but it could equally well have come from a database
- * or received through another LANSa Integrator service call

```

c          eval   xllistptr = %addr(xldata)

```

- * Write the list of items from the compile-time array to the specified
- * position in the specified worksheet:
- * - this passes the multiple occurrence data structure
- * (xllist) containing the items
- * - the structure is described to the ExcelReadService by the
- * structure XML identified by the SERVICE_STRUCTURE keyword - th
- * must be a matching entry in the structure.properties file and a
- * corresponding structure XML file

- * NOTE: this call uses the JSMCMDX api in order to be able to send
- * variable data (in this case the structure/list)

```

c          eval   jsmcmd = 'WRITE'
c                + ' SHEET(ORDERLINES) R1C1(5,3)'
c                + ' SERVICE_STRUCTURE(XL.XLOrderLine)'
c                + ' COUNT(' + %char(xlocur) + ')'
c                + ' OCCURS(' + %char(xlocur) + ')'
c                + ' SIZE(' + %char(xlsize) + ')'

```

```

c          eval   bytelength = xlocur * xlsize

```

```

c          callp  p_jsmcmdx(jsmcmd:xllist:bytelength:
c                jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg20:*COMP')

```

* Close the Excel workbook

```
c          eval    jsmcmd = 'CLOSE'
c          callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg30:*COMP')
```

* Unload the ExcelReadService

```
c          eval    jsmcmd = 'SERVICE_UNLOAD'
c          callp   p_jsmcmd(jsmcmd:jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg98:*COMP')
```

* Close the connection to the JSM server and finish

```
c          callp   p_jsmclose(jsmsts:jsmmsg)
c          callp   CheckResult(jsmsts:jsmmsg)
c          callp   SendMessage(CompMsg99:*COMP')
c          eval    *inlr = *on
c          return
```

* Procedure to write a single cell at a time to the workbook

```
p WriteCell    b
d WriteCell    pi
d wcR1C1          10a value
d wcCell          256a value

c          eval    jsmcmd = 'WRITE'
c          + ' R1C1(' + %trim(wcR1C1) + ')'
c          + ' SHEET(ORDERLINES)'
c          + ' SERVICE_STRUCTURE(XL.XLCell)'
c          + ' COUNT(1) OCCURS(1)'
c          + ' SIZE(' + %char(%size(wcCell)) + ')'

c          eval    bytelength = %size(wcCell)
```

```

c          callp  p_jsmcmdx(jsmcmd:wcCell:bytlength:
c                      jsmsts:jsmmsg)
c          callp  CheckResult(jsmsts:jsmmsg)
c          callp  SendMessage(CompMsg20:*'COMP')

```

```

p WriteCell      e

```

```

*****
* Procedure to check the result of a Java Service Manager call
*****

```

```

p CheckResult    b
d CheckResult    pi
d crjsts          const like(jsmsts)
d crjmsg         const like(jsmmsg)
d crText         s      512a
d crMsg1         c      const('JSM Status : ')
d crMsg2         c      const('JSM Message: ')
d crMsg3         c      const('JSM Service error has +
d                  occurred')
c          if     crjsts <> 'OK'
c          eval   crText = crMsg1 + crjsts
c          callp  SendMessage(crText:*'DIAG')
c          eval   crText = crMsg2 + crjmsg
c          callp  SendMessage(crText:*'DIAG')
c          callp  SendMessage(crMsg3:*'ESCAPE')
c          endif
p CheckResult    e

```

```

*****
* Procedure to send a program message
*****

```

```

p SendMessage    b
d SendMessage    pi
d smText         512a  VALUE
d smMsgT         10a  VALUE
d smMsgI         s      7a  inz('CPF9897')
d smMsgF         s      20a  inz('QCPFMSG *LIBL  ')

```

```
d smDtaL      s      10i 0 inz(%size(smText))
d smStkE      s      10a  inz('*')
d smStkC      s      10i 0 inz(1)
d smMsgK      s      4a
d smErrC      s      10i 0 inz(0)
```

```
c      if      smMsgT = '*ESCAPE'
c      eval    smMsgI = 'CPF9898'
c      endif
```

```
c      call    'QMHSNDPM'
c      parm    smMsgI
c      parm    smMsgF
c      parm    smText
c      parm    smDtaL
c      parm    smMsgT
c      parm    smStkE
c      parm    smStkC
c      parm    smMsgK
c      parm    smErrC
```

```
p      e
**CTDATA xldata
0000001123  Gasket Paper      0000009950000010
0000002456  Gasket polymer glue      0000013270000005
```

ILE RPG Client JSM API Example

The following example shows how the JSM APIs are used from an ILE RPG program. This example is not a complete RPG program.

```
D JSMSRV          S   50
D JSMSTS          S   20
D JSMMMSG         S  255
D JSMCMD          S  255
D BYTEDATA        S 32767
D BYTELENGTH     S 10I 0

C  MOVE  *BLANKS  JSMSRV
C  MOVE  *BLANKS  JSMSTS
C  MOVE  *BLANKS  JSMMMSG
C  MOVE  *BLANKS  JSMCMD
C  MOVE  *BLANKS  BYTEDATA
C  Z-ADD 32767    BYTELENGTH

*
* Open connect to default JSM server
*
C  CALLB 'JSMOPEN'
C  PARM                JSMSRV
C  PARM                JSMSTS
C  PARM                JSMMMSG

*
* Send command to service and receive status and response
*
C  MOVEL *BLANKS  JSMCMD

*
C  CALLB 'JSMCMD'
C  PARM  'MYCMD'  JSMCMD
C  PARM                JSMSTS
C  PARM                JSMMMSG

*
* Send command and variable data to service
```

*

C MOVEL 'MYCMD' JSMCMD

*

C CALLB 'JSMCMDX'

C PARM JSMCMD

C PARM BYTEDATA

C PARM BYTELENGTH

C PARM JSMSTS

C PARM JSMMSG

*

* Close service

*

C CALLB 'JSMCLOSE'

C PARM JSMSTS

C PARM JSMMSG

*

C SETON LR

C Examples

[C Client Open/Close Example](#)

[C Client Working List Example](#)

C Client Open/Close Example

```
#include <stdio.h>
#include <stdlib.h>

/* JSM header */
#include "x_jsm.h"

#define JSMHOST "LOCALHOST:4560"

static char status[JSM_STATUS_SIZE+1];
static char response[JSM_RESPONSE_SIZE+1];

/* Main function */

int main(int argc, char **argv);

/* Utility functions */

void trim(char *str);
void printCharArray(char *chars, int size);

int main(int argc, char **argv)
{
    char *host;

    /* Get host from command line */

    if (argc > 1)
    {
        host = argv[1];
    }
    else
    {
        host = JSMHOST;
    }
}
```

```
}
```

```
/* Open JSM connection */
```

```
printf ( "\nJSMOPEN %s\n" , host );
```

```
JSMOPEN ( host , status , response );
```

```
trim ( response );
```

```
printf ( "status = %s\n" , status );
```

```
printf ( "response = %s\n\n" , response );
```

```
/* Check status */
```

```
if ( strcmp ( status , "OK" ) != 0 )
```

```
{
```

```
    return 1;
```

```
}
```

```
/* Close JSM connection */
```

```
printf ( "JSMCLOSE\n" );
```

```
JSMCLOSE ( status , response );
```

```
trim ( response );
```

```
printf ( "status = %s\n" , status );
```

```
printf ( "response = %s\n\n" , response );
```

```
return 0;
```

```
}
```

```
void trim(char *str)
```

```
{
```

```
    char *end = str;
```

```
    while (*str)
```

```
    if (*(str++) != ' ')
        end = str;
    *end = '\0';
}
```

```
void printCharArray(char *chars, int size)
{
    int i;
    putchar("");
    for (i=0; i<size; i++)
        if (*chars)
            putchar(*(chars++));
        else
            break;
    putchar("");
    putchar('\n');
}
```

C Client Working List Example

```
#include <stdio.h>
#include <stdlib.h>

/* JSM header */
#include "x_jsm.h"

#define JSMHOST "LOCALHOST:4560"

#define MAX_BYTE_LENGTH 32767
#define WORKING_LIST_SIZE 100
#define PK1_LENGTH 10
#define PK2_LENGTH 7

/* Working list entry definition:
 *
 * <rdml:field name="TXT" type="A" length="30" />
 * <rdml:field name="PK1" type="P" length="10" decimal="2" />
 * <rdml:field name="PK2" type="P" length="7" decimal="0" />
 *
 * Bytes for packed field = ( length / 2 ) + 1
 */

struct Example3Line
{
    char txt[30];
    char pk1[(PK1_LENGTH / 2) + 1];
    char pk2[(PK2_LENGTH / 2) + 1];
};

static char status[JSM_STATUS_SIZE+1];
static char response[JSM_RESPONSE_SIZE+1];
static char byteArray[MAX_BYTE_LENGTH];
static int byteArrayLength;

/* Simple wrapper functions for JSM API calls */
```

```

void jsmOpen(char *cmd);
void jsmCommand(char *cmd);
void jsmCommandX(char *cmd, void *target, int strucSize);
void jsmClose();

/* Main function */

int main(int argc, char **argv);

void example3Test();

/* Utility functions */

struct Example3Line makeEx3Line(char *txt, long pk1, long pk2);
void trim(char *str);
void printCharArray(char *chars, int size);
void printEx3Line(struct Example3Line line);
void getByteArray(int *byteSize, char *byteArray);
void longToPacked(unsigned char *packed, long num, int length);
long packedToLong(unsigned char *packed, int length);

void jsmOpen(char *cmd)
{
    printf("\nJSMOPEN %s\n",cmd);
    JSMOPEN(cmd,status,response);
    trim(response);
    printf("status = %s\nresponse = %s\n",status,response);
}

void jsmCommand(char *cmd)
{
    printf("\n%s\n",cmd);
    JSMCMD(cmd,status,response);
    trim(response);
    printf("status = %s\nresponse = %s\n",status,response);
}

```

```
void jsmCommandX(char *cmd, void *target, int structSize)
{
    printf("\n%s\n",cmd);
    byteArrayLength = structSize;
    JSMCMDX(cmd,(char*)target,&byteArrayLength,status,response);
    trim(response);
    printf("status      = %s\nresponse      = %s\n",status,response);
}
```

```
void jsmClose()
{
    printf("\nJSMCLOSE\n");
    JSMCLOSE(status,response);
    trim(response);
    printf("status  = %s\nresponse = %s\n",status,response);
}
```

```
int main(int argc, char **argv)
{
    if (argc > 1)
    {
        jsmOpen(argv[1]);
    }
    else
    {
        jsmOpen(JSMHOST);
    }
}
```

```
example3Test();
```

```
jsmClose();
```

```
return 0;
}
```

```
void example3Test()
{
```

```

char command[JSM_COMMAND_SIZE+1];
int byteSize;
int i;

struct Example3Line workingList[WORKING_LIST_SIZE];
struct Example3Line fields = makeEx3Line( "3GL client field",
                                          -987654321,
                                          180980 );

workingList[0] = makeEx3Line( "Text line 1", 12345, 678 );
workingList[1] = makeEx3Line( "Text line 2", -52300, -3918 );

jsmCommand("SERVICE_LOAD SERVICE(EXAMPLE3)");

sprintf ( command,
          "SETFIELDS SERVICE_STRUCTURE(EXAMPLE3.LINE) SIZE(%d",
          sizeof(fields)
          );

jsmCommandX(command,&fields,sizeof(fields));
getByteArray(&byteSize,(char*)&fields);

printf("\nFields\n");
printEx3Line(fields);

sprintf ( command,
          "ADDEENTRY SERVICE_STRUCTURE(EXAMPLE3.LINE) SIZE(%d",
          sizeof(fields), WORKING_LIST_SIZE, 2
          );

jsmCommandX(command,workingList,sizeof(workingList));
getByteArray(&byteSize, (char*) workingList);

for (i=0; i<(byteSize/sizeof(struct Example3Line)); i++)
{
    printf("\nWorking list entry %d\n",i+1);
    printEx3Line(workingList[i]);
}
}

```

```

struct Example3Line makeEx3Line(char *txt, long pk1, long pk2)
{
    struct Example3Line line;
    memset(line.txt, ' ', 30);
    memcpy(line.txt, txt, strlen(txt));
    longToPacked(line.pk1, pk1, PK1_LENGTH);
    longToPacked(line.pk2, pk2, 7);
    return line;
}

```

```

void trim(char *str)
{
    char *end = str;
    while (*str)
        if (*(str++) != ' ')
            end = str;
    *end = '\0';
}

```

```

void printCharArray(char *chars, int size)
{
    int i;
    putchar("");
    for (i=0; i<size; i++)
        if (*chars)
            putchar(*(chars++));
        else
            break;
    putchar("");
    putchar('\n');
}

```

```

void printEx3Line(struct Example3Line line)
{
    printf("  TXT = ");
    printCharArray(line.txt, 30);
    printf("  PK1 = %.2f\n", packedToLong(line.pk1, PK1_LENGTH)/100.0);
}

```

```
printf("  PK2 = %d\n",packedToLong(line.pk2,PK2_LENGTH));  
}
```

```
void getByteArray(int *byteSize, char *byteArray)  
{  
    JSMBYTELENGTH(byteSize);  
    printf("byte array size = %d\n",*byteSize);  
    JSMBYTERECV(byteArray);  
}
```

```
void longToPacked(unsigned char *packed, long num, int length)  
{  
    int packedLen = (length / 2) + 1;  
    int i;
```

```
    unsigned char *c = &packed[packedLen-1];
```

```
    /* Sign */  
    if (num < 0)  
    {  
        *c = 0x0D;  
        num = -num;  
    }  
    else  
    {  
        *c = 0x0F;  
    }  
}
```

```
    /* High nibble of last byte */  
    *c |= (num % 10) << 4;  
    num /= 10;
```

```
    for (c = &packed[packedLen-2]; c >= packed; c--)  
    {  
        *c = num % 10;  
        num /= 10;  
        *c |= (num % 10) << 4;  
        num /= 10;  
    }  
}
```

```
}
```

```
long packedToLong(unsigned char *packed, int length)
```

```
{
```

```
int packedLen = (length / 2) + 1;
```

```
long num = 0;
```

```
unsigned char hi,lo;
```

```
int i;
```

```
for (i=0; i<packedLen; i++)
```

```
{
```

```
hi = (packed[i] & 0xF0) >> 4;
```

```
lo = (packed[i] & 0x0F);
```

```
if (i < (packedLen-1))
```

```
{
```

```
if (hi > 9 || lo > 9)
```

```
return 0;
```

```
num = (num * 100) + (hi * 10) + lo;
```

```
}
```

```
else
```

```
{
```

```
num = (num * 10) + hi;
```

```
if (lo == 0x0D)
```

```
num = -num;
```

```
else if (lo != 0x0F)
```

```
num = 0;
```

```
}
```

```
}
```

```
return num;
```

```
}
```

Appendix C. EDI Services

Electronic Data Interchange (EDI) is a set of standards for structuring information that is to be electronically exchanged between and within businesses, organizations, government entities and other groups.

The EDI standards were designed to be independent of communication and software technologies.

EDI can be transmitted using any methodology agreed to by the sender and recipient.

This includes a variety of technologies such as FTP, SMTP, HTTP, AS1, AS2 and AS3.

LANSA's EDI facility consists of these services:

[EDICatalogService](#)

[EDIDataSetService](#)

[EDIDatabaseService](#)

[EDIInterchangeService](#)

EDICatalogService

Service Name: EDICatalogService

The EDICatalogService is used to parse, validate and process EDI documents.

The EDICatalogService supports the following commands:

SERVICE_LOAD

SERVICE_GET

DELETE

RENAME

COPY

MOVE

LIST

SET

GET

READ

SPLIT

BREAK

SPLITBREAK

FORMAT

VALIDATE

SERVICE_UNLOAD

SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD

SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

DELETE

The DELETE command deletes the specified file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
DELETE	FILE	value	Mandatory. File Path.

RENAME

The RENAME command renames the specified local file to a new name. The TO file must be in the same directory as the original, FROM file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RENAME	FROM	value	Mandatory. Old file name.
	TO	value	Mandatory. New file name.
	REPLACE	*YES	Optional. Replace file.
		*NO	Do not replace file. The default value of the REPLACE keyword is *NO. If the TO filename exists, then the rename process is stopped, unless the REPLACE keyword is *YES.

COPY

The COPY command copies the specified local file to a new file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RENAME	FROM	value	Mandatory. Old file name.
	TO	value	Mandatory. New file name.
	REPLACE	*YES	Optional. Replace file.
		*NO	Do not replace file. The default value of the REPLACE keyword is *NO. If the TO filename exists, then the rename process is stopped, unless the REPLACE keyword is *YES.

MOVE

The MOVE command moves the specified local file to a new file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RENAME	FROM	value	Mandatory. Old file name.
	TO	value	Mandatory. New file name.
	REPLACE	*YES	Optional. Replace file.
		*NO	Do not replace file. The default value of the REPLACE keyword is *NO. If the TO filename exists, then the rename process is stopped, unless the REPLACE keyword is *YES.

LIST

The LIST command will fill the command's list object with absolute file names. A single field working list is required to receive the canonical file paths.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
LIST	DIR	value	Optional. Fill working list with directory listing. All files in that directory will be selected.
	EXT	value	Optional. Filtering on file extension. The filtering match is case insensitive.

If a directory path specified by the DIR keyword is used then filenames in that directory will be selected.

SET

The SET command sets the current working directory.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SET	DIR	value	Mandatory. Directory.
-----	-----	-------	-----------------------

GET

The GET command returns the current validation errors into the specified working list.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

GET	OBJECT	*ERROR	Mandatory.
-----	--------	--------	------------

READ

The READ command reads and parses the specified EDI file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
READ	FILE	value	Conditional. File path.
	ENCODING	value	Optional. File encoding. Default. ISO8859_1.
	IGNORE	*NONE	Optional. Ignore characters. Default. *NONE
		*CRLF	Ignore CR and LF characters.

The READ command reads the parsed interchanges for the specified information.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
READ	DATA	*TA1	Optional.
		*TA3	Optional.
		*INTERCHANGE	Optional.
		*GROUP	Optional.
		*TRANSACTION	Optional.

SPLIT

The SPLIT command splits the specified EDI file into multiple files.

Each file contains a single interchange and the each new file uses the base name of the EDI file and a sequence number.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SPLIT	FILE	value	Conditional. File path.
	ENCODING	value	Optional. File encoding. Default. ISO8859_1.
IGNORE		*NONE	Optional. Ignore characters. Default. *NONE
		*CRLF	Ignore CR and LF characters.
KEEP		*YES	Optional. Keep file.
		*NO	Default.

BREAK

The BREAK command breaks the specified EDI file into multiple files.

Each file contains a specified number of transactions and the each new file uses the base name of the EDI file and a sequence number.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
BREAK	FILE	value	Conditional. File path.
	ENCODING	value	Optional. File encoding. Default. ISO8859_1.
	IGNORE	*NONE	Optional. Ignore characters. Default. *NONE
		*CRLF	Ignore CR and LF characters.
KEEP		*YES	Optional. Keep file.
		*NO	Default.
FORMAT		*NONE	Optional. Default. *NONE.
		*CRLF	
LIMIT		number	Optional. Default. 2.

SPLITBREAK

The SPLITBREAK command combines features from the SPLIT and BREAK commands.

Each file contains a specified number of transactions and the each new file uses the base name of the EDI file and a sequence number.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SPLITBREAK	FILE	value	Conditional. File path.
	ENCODING	value	Optional. File encoding. Default. ISO8859_1.
	IGNORE	*NONE	Optional. Ignore characters. Default. *NONE
		*CRLF	Ignore CR and LF characters.
KEEP	*YES	Optional. Keep file.	
	*NO	Default.	
FORMAT	*NONE	Optional. Default. *NONE.	
	*CRLF		
LIMIT		number	Optional. Default. 2.

FORMAT

The FORMAT command operates at the byte stream level and is used to remove and add CR and LF characters from the EDI byte stream.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
FORMAT	FROM	value	Mandatory. File path.
	TO	value	Mandatory. File path.
		*FROM	
BLOCK		number	Conditional.
		*NOMAX	Default.
SEPARATOR		value	Conditional. If no separator is specified then block size is used.

VALIDATE

The VALIDATE command is used to validate a specified transaction.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
VALIDATE	SEF	value	Mandatory. file path.
	INTERCHANGE	value	Mandatory. Interchange.
	GROUP	value	Conditional. Group. Default. blank.
	TRANSACTION	value	Mandatory. Transaction.

SERVICE_UNLOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

EDIDataSetService

Service Name: EDIDataSetService

The EDIDataSetService is used to create a DataSet from an EDI transaction and to create an EDI transaction from a DataSet.

The EDIDataSetService supports the following commands:

SERVICE_LOAD

SERVICE_GET

GET

READ

WRITE

SERVICE_UNLOAD

SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD

SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

GET

The GET command returns the current validation errors into the specified working list.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

GET	OBJECT	*ERROR	Mandatory.
-----	--------	--------	------------

READ

The READ command reads the specified transaction and creates a DataSet file.

Syntax:

Command Keyword	Value	Developer notes for Command/Keyword/Value
READ	FILE	value Mandatory. File path
	SEF	value Mandatory. File path.
	MAP	value Mandatory. File path.
	DATASET	value Mandatory. File path,
	ENCODING	value Optional. File encoding. Default. ISO8859_1.
	IGNORE	*NONE Optional. Default. *NONE.
		*CRLF
	MAPTRACE	*YES Optional. Trace mapping.
		*NO Default.
	INTERCHANGE	value Mandatory. Interchange.
	GROUP	value Conditional. Group. Default. blank.
	TRANSACTION	value Mandatory. Transaction.

WRITE

The WRITE command reads the specified DataSet file and creates a transaction file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword	
WRITE	FILE	value	Mandatory. File path	
	SEF	value	Mandatory. File path	
	MAP	value	Mandatory. File path	
	DATASET	value	Mandatory. File path	
	ENCODING	value	Optional. File encoding Default. ISO8859_1.	
	DOCUMENT	*X12		Optional. Default. *X12.
		*EDIFACT		
		*TRADCOM		
	COMPRESS	*YES		Optional. Default. *YES.
		*NO		
SEGMENT_DELIMITER	value	Optional.		
ELEMENT_DELIMITER	value	Optional.		
COMPOSITE_DELIMITER	value	Optional.		
DECIMAL	value	Optional.		
ESCAPE	value	Optional.		

SERVICE_UNLOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

EDIDatabaseService

Service Name: EDIDatabaseService

The EDIDatabaseService is used to send a DataSet to a database and to receive a DataSet from a database.

The EDIDatabaseService supports the following commands:

SERVICE_LOAD

SERVICE_GET

CONNECT

SET

SEND

RECEIVE

COMMIT

ROLLBACK

METADATA

DISCONNECT

SERVICE_UNLOAD

SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD

SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

CONNECT

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
CONNECT	DATABASE	value	Mandatory.
	DRIVER	value	Mandatory.
	USER	value	Optional.
	PASSWORD	value	Optional.

SET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword
SET	ONERROR	*ROLLBACK	Optional.
		*DISCONNECT	
		*CONTINUE	
	READONLY	*YES	Optional.
		*NO	
	AUTOCOMMIT	*YES	Optional.
		*NO	
	ISOLATION	*NONE	Optional.
		*READCOMMITTED	
		*READUNCOMMITTED	
		*REPEATABLEREAD	
		*SERIALIZABLE	

SEND

The SEND command updates the current database using the specified DataSet file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	DATASET	value	Mandatory. File path.
	TRADERID	value	Optional.
	IDENTIFIER	value	Optional.
	MODE	*STREAM	Optional.
		*DATASET	Default.

RECEIVE

The RECEIVE command creates a DataSet file using the current database connection and specified map.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	MAP	value	Mandatory. File path.
	TRADERID	value	Optional.
	IDENTIFIER	value	Optional.
	DATASET	value	Mandatory. File path

COMMIT

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

COMMIT

ROLLBACK

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
---------	---------	-------	--

ROLLBACK

METADATA

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
METADATA	TABLE	value	Mandatory. Database Table.

DISCONNECT

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

DISCONNECT			
------------	--	--	--

SERVICE_UNLOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

EDIInterchangeService

Service Name: EDIInterchangeService

The EDIInterchangeService is used to collate intermediate transactions into a single interchange document.

The EDIInterchangeService supports the following commands:

SERVICE_LOAD

SERVICE_GET

BEGIN

ADD

END

WRITE

VALIDATE

GET

SERVICE_UNLOAD

SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD

SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

BEGIN

The BEGIN command begins a new interchange.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword
BEGIN	SEF	value	Mandatory. File path.
	INTERCHANGE	*X12 *EDIFACT *TRADCOM	Mandatory.
	SEGMENT_DELIMITER	value	Optional.
	ELEMENT_DELIMITER	value	Optional.
	COMPOSITE_DELIMTER	value	Optional.
	DECIMAL	value	Optional.
	ESCAPE	value	Optional.
	ENCODING	value	Optional. Default. ISO8859_1.
	IDENTIFIER	value	Optional.

ADD

The ADD command adds groups and transactions to current interchange.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
ADD	OBJECT	*TA1 *TA3 *GROUP *TRANSACTION	Mandatory.
	IDENTIFIER	value	Optional.
	FILE	value	Conditional. Transaction file.

END

The END command closes the current interchange.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

END

WRITE

The WRITE command writes the current interchange out to the specified file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
WRITE	FILE	value	Manadatory. File path.
	APPEND	*YES	Optional.
		*NO	Default.

VALIDATE

The **VALIDATE** command is used to validate the current interchange.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
VALIDATE	SEF	value	Mandatory. file path.
	INTERCHANGE	value	Mandatory. Interchange.
	GROUP	value	Conditional. Group. Default. blank.
	TRANSACTION	value	Mandatory. Transaction.

GET

The GET command returns the current validation errors into the specified working list.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

GET	OBJECT	*ERROR	Mandatory.
-----	--------	--------	------------

SERVICE_UNLOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

Appendix D. AS2 and AS3 Services

AS2 and AS3 are a security standard defined by the Working Group of the Internet Engineering Task Force (IETF) that allows EDIFACT, X12 and XML-based business transactions to move securely over the Internet.

The standard underlines the critical elements of data privacy, data authentication and non-repudiation of origin and receipt, required to ensure the integrity of data communications via the Internet.

The business benefits of AS2 and AS3 interoperability include boosting efficiency and lowering the cost for transmitting critical business data.

LANSA's AS2 and AS3 facility consists of these services:

[AS2InboundService](#)

[AS2OutboundService](#)

[AS2OutboundReceiptService](#)

[AS2CertificateService](#)

[AS3FileService](#)

AS2InboundService

Service Name: AS2InboundService

The AS2InboundService is used to receive an AS2 transaction and send an appropriate response.

The AS2InboundService supports the following commands:

SERVICE_LOAD

SERVICE_GET

GET

RECEIVE

SAVE

SEND

SERVICE_UNLOAD

SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD

SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

GET

The GET command is used to access information from the received transaction.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	PROPERTY	value	Mandatory. Property keyword.

RECEIVE

The RECEIVE command is used to receive an AS2 transaction.

The following keywords may be supplied on the RECEIVE command or as an entry in a keyword/value list argument.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	RECEIVER	value	Mandatory. The identity that receives the transaction. This value is used in resource file lookups. Value is made up of 2 components separated by a colon, a name component and an optional private keystore password component. Example: ACME:mypassword
	PARTNER	value	Optional. Authentication partner.

The RECEIVER keyword value is used to obtain the decryption key file, password, decryption certificate and verify authentication certificate using a selection hierarchy of entries from the properties file.

keystore.{receiver}= pki/filename

keystore = pki/filename

keystore.password.{receiver} = password

keystore.password = password

certificate.{receiver} = pki/filename

certificate = pki/filename

authentication.certificate.{partner}=pki/filename

The following properties come from the received HTTP request.

Property	Description	Values

*AS2TO	AS2-To.	
*AS2FROM	AS2-From.	
*AS2VERSION	AS2- Version.	
*FROM	From.	
*DATE	Date.	
*SUBJECT	Subject.	
*MESSAGEID	Message-Id.	
*RECEIPTDELIVERY	Receipt- Delivery- Option.	The delivery address for an asynchronous receipt. Blank if synchronous receipt requested.
*RECEIPTPROTOCOL	Receipt- Delivery- Option.	If asynchronous receipt is required, this contains the protocol required by the receipt. HTTP
value	HTTP protocol property.	

The following properties come from the received transaction.

Property	Description	Example message values
*MICMD5	Calculated MD5 MIC.	
*MICSHA1	Calculated SHA1 MIC.	
*MICSHA256	Calculated SHA256 MIC.	
*MICSHA384	Calculated	

	SHA384 MIC.	
*MICSHA512	Calculated SHA512 MIC.	
*CONTENT	Transaction data content type.	*XML or *X12 or *EDIFACT or *RECEIPT or mime type in uppercase.
*CONTENTTYPE	Transaction content type in full.	application/xml; name=xxxxxxxxxxxx

The following properties come from the received receipt.

Use the GET PROPERTY (*CONTENT) to check if the received transaction is a receipt.

Property	Description	Example message values
*RECEIPTMESSAGEID	Message ID of the original transaction. This can be used to match to the original sent transaction.	
*RECEIPTMIC	MIC of the original transaction.	
*RECEIPTMICALG	MIC algorithm.	MD5, SHA1
*RECEIPTDISPOSITIONTYPE	Status of the original AS2 transaction.	Possible values: processed displayed dispatched deleted

		denied failed.
*RECEIPTDISPOSITIONMODIFIER	Status of the original AS2 transaction.	Possible values: "" error warning superseded expired mailbox-terminated.

SAVE

The SAVE command is used to save the received transaction or receipt.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SAVE	FILE	value	Mandatory. Filename
------	------	-------	---------------------

SEND

The SEND command is used to return a response to the AS2 client program. The SEND command internally uses the Disposition-Notification-To and the Disposition-Notification-Options received from the client program to help in automatic creation of a response.

If a synchronous receipt was requested, a receipt must be sent.

If an asynchronous receipt was requested, a basic 200 OK response is sent.

Note: For asynchronous receipts, the receipt may be sent later to the requested address. The Receipt-Delivery-Option contains the delivery URL.

The list argument will be returned with the following keywords and values:

Keyword	Description
MESSAGEID	Message-Id of the receipt sent.
SUBJECT	Subject on the receipt.
FROM	From value on the receipt.
DATE	Date on the receipt.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	RESPONSE	*DEFAULT	Mandatory. The type of response is automatically determined from the client request. Either basic, receipt or signed receipt is returned.
		*OK	This will send a basic OK (200 HTTP response). This response should only be used if an asynchronous receipt was requested.

	*ERROR	Send an error receipt.
FROM	value	Optional. From.
SUBJECT	value	Optional. Subject.
TEXT	value	Optional. Text.
PROCESSED	value	Optional. Default value is "processed".
MODIFER	value	Optional. Default value is blank or "error" for response *ERROR.
MESSAGEID	value	Optional. Complete transaction message id. The Message-Id is normally generated.

The RECEIVER keyword value is used to create the following receipt information.

receipt.from.{receiver} = email address

receipt.from = email address

receipt.subject.{receiver} = subject

receipt.subject = subject

messageid.prefix.{recipient} = company identifier

messageid.prefix. = company identifier

Human readable text is obtained from files using a properties file selection hierarchy.

receipt.message.{receiver} = system/receipt-message.txt

receipt.message = system/receipt-message.txt

receipt.text.{receiver} = system/receipt-text.txt

receipt.text = system/receipt-text.txt

receipt.error.{receiver} = system/receipt-error.txt

receipt.error = system/receipt-error.txt

The receipt.message contains the human readable component of the receipt.

The receipt.text contains the text that is sent as the body content in a 200 OK basic response. It is sent as content type text/plain.

The receipt.error contains the human readable component of the error receipt.

The following special merge labels are supported in the receipt.text, receipt.message and receipt.error files:

%%COMPANYFROM%%

%%COMPANYTO%%

%%DATE%%

%%SUBJECT%%

%%TEXT%%

Sample receipt.message file content:

The message sent to recipient <%%COMPANYTO%%> on %%DATE%% with

Sample receipt.text file content:

Data from <%%COMPANYFROM%%> has been received by <%%COMPAN

Sample receipt.error file content:

The message sent

to recipient <%%COMPANYTO%%> on %%DATE%% with subject "%%SU
following error %%TEXT%%.

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

DO NOT use the SERVICE_UNLOAD command when sending a HTTP response. Use the SEND command to create the response and then use the JSM_CLOSE BIF to close the connection. The byte array response returned to JSM Client program by the SEND command will be become the HTTP response.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

Sample AS2InboundService resources

```
#
# AS2InboundService resource ( Default )
#
messageid.prefix=DEFAULT-AS2
keystore=pki/default-key.p8
keystore.password=password
certificate=pki/default-cert.cer
receipt.from=as2manager@company.com
receipt.subject=Your request MDN response (default)
receipt.text=system/default-receipt-text.txt
receipt.message=system/default-receipt-message.txt
#
# Receiver – ACME
#
messageid.prefix.acme=ACME-AS2
keystore.acme=pki/acme-key.p8
keystore.password.acme=password
certificate.acme=pki/acme-cert.cer
receipt.from.acme=as2manager@acme.com.au
receipt.subject.acme=Your request MDN response
receipt.text.acme=system/acme-receipt-text.txt
receipt.message.acme=system/acme-receipt-message.txt
#
message.001=Service successfully loaded
message.002=Service successfully unloaded
message.003=Command is not supported :
#
```

AS2OutboundService

Service Name: AS2OutboundService

The AS2OutboundService is used to send an AS2 transaction.

The AS2OutboundService supports the following commands:

[SERVICE_LOAD](#)

[SERVICE_GET](#)

[SEND](#)

[GET](#)

[SERVICE_UNLOAD](#)

SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD

SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

SEND

The SEND command is used to send an AS2 transaction.

The following keywords may be supplied on the SEND command or as an entry in a keyword/value list argument.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	RECIPIENT	value	Mandatory. The identity that that will receive the transaction. This value is used in resource file lookups. Value is made up of 2 components separated by a colon, a name component and an optional private keystore password component. Example: ACME:mypassword
	FILE	value	Mandatory. Filename of content.
	NAME	value	Optional. Override name of filename used in message.
	CONTENT	value	Mandatory. Content mime-type.
		*XML	application/xml.
		*X12	application/edi-x12.
		*EDIFACT	application/edifact.
	MESSAGEID	value	Optional. Complete transaction message id. The Message-Id is normally generated.
	SUBJECT	value	Optional. Transaction subject.
	RECEIPTTO	value	Optional. Require receipt.

	*NONE	Do not send Disposition-Notification-To.
RECEIPTSIGNED	*YES	Optional. Require signed receipt.
	*NO	
DELIVERY	value	Optional. URL for asynchronous receipt delivery.
	*NONE	Do not send Receipt-Delivery-Option.
SIGNED	*YES	Optional. Signed Transaction.
	*NO	Default.
ENCRYPTED	*YES	Optional. Encrypted Transaction.
	*NO	Default.
COMPRESSED	*YES	Optional. Compressed Content.
	*NO	Default.
USER	value	Optional. Basic authentication user.
PASSWORD	value	Optional. Basic authentication password.
PARTNER	value	Optional. Authentication partner.

The RECIPIENT keyword value is used to obtain the signing key file, password, signing certificate, encryption certificates and verify authentication of signed receipt certificate using a selection hierarchy of entries from the properties file.

keystore.{recipient}= pki/filename

keystore = pki/filename

keystore.password.{recipient} = password
keystore.password = password
certificate.{recipient} = pki/filename
certificate = pki/filename
encryption.certificates.{recipient} = pki/filename, pki/filename, pki/filename
authentication.certificate.{partner} = pki/filename
url.{recipient} = http://company.com/as2server
as2-to.{recipient} = companyto
as2-from.{recipient} = companyfrom
as2-from = companyfrom
from.{recipient} = from
from = from
subject.{recipient} = subject
subject = subject
receipt.to.{recipient} = *none | email address (Disposition-Notification-To)
receipt.to = *none | email address (Disposition-Notification-To)
receipt.signed.{recipient} = *yes | *no (Disposition-Notification-Options)
receipt.signed = *yes | *no (Disposition-Notification-Options)
receipt.delivery.{recipient} = *none | url (Receipt-Delivery-Option)
receipt.delivery = *none | url (Receipt-Delivery-Option)
messageid.prefix.{recipient} = company identifier
messageid.prefix. = company identifier
signed.algorithm.{recipient} = *MD5 | *SHA1 | *SHA256 | *SHA384 |
*SHA512
signed.algorithm = *MD5 | *SHA1 | *SHA256 | *SHA384 | *SHA512
encryption.algorithm.{recipient} = *DES3 | *AES128 | *AES192 | *AES256
encryption.algorithm = *DES3 | *AES128 | *AES192 | *AES256

GET

The GET command is used to access information from the sent transaction and the HTTP response from the remote AS2 server.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	PROPERTY	value	Mandatory. Property keyword.

The following properties come from the created data source that was sent.

Property	Description	Values
*DATAMESSAGEID	Message-Id.	
*DATAMICMD5	Calculated MD5 MIC.	
*DATAMICSHA1	Calculated SHA1 MIC.	
*DATAMICSHA256	Calculated SHA256 MIC.	
*DATAMICSHA384	Calculated SHA384 MIC.	
*DATAMICSHA512	Calculated SHA512 MIC.	
*DATARECEIPT	Receipt requested.	*YES or *NO
*DATARECEIPTSIGNED	Signed receipt requested.	*YES or *NO.
*DATARECEIPTDELIVERY	Receipt delivery requested.	*YES = asynchronous *NO = synchronous

The following properties come from the HTTP response.

Property	Description	Values
*AS2TO	AS2-To.	
*AS2FROM	AS2-From.	
*AS2VERSION	AS2-Version.	
*FROM	From.	
*DATE	Date.	
*SUBJECT	Subject.	
*MESSAGEID	Message-Id.	
*CONTENT	Type of response content.	*RECEIPT or *OTHER
value	HTTP protocol property.	

The following properties come from the received receipt.

Use the GET PROPERTY (*CONTENT) to check if the received response is a receipt.

Property	Description	Example message values
*RECEIPTMESSAGEID	Message ID of the original transaction. This can be used to match to the original sent transaction.	
*RECEIPTMIC	MIC returned in receipt. This can be compared with the	

	original transaction MIC to check the transaction delivery.	
*RECEIPTMICALG	MIC algorithm.	MD5, SHA1
*RECEIPTDISPOSITIONTYPE	Status of the AS2 transaction.	Possible values: processed displayed dispatched deleted denied failed.
*RECEIPTDISPOSITIONMODIFIER	Status of the AS2 transaction.	Possible values: "" error warning superseded expired mailbox- terminated.

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

Sample AS2OutboundService resources

```
#
# AS2OutboundService resource ( Default )
#
messageid.prefix=DEFAULT-AS2
subject=Default subject
from=default@company.com
as2-from="Default Company"
keystore=pki/default-key.p8
keystore.password=password
certificate=pki/default-cert.cer
# receipt.signed=*yes | *no
receipt.to=notify@company.com
#
# Recipient – ACME
#
messageid.prefix.acme=ACME-AS2
as2-to.acme="Acme Corporation"
as2-from.acme="Rocket Inc"
from.acme=person@rocket.com.au
url.acme=http://www.acme.com.au/cgi-bin/jsmdirect?as2server
subject.acme=The subject
keystore.acme=pki/acme-key.p8
keystore.password.acme=password
certificate.acme=pki/acme-cert.cer
encryption.certificates.acme=pki/acme-cert.cer,pki/default-cert.cer
receipt.to.acme=notify@rocket.com.au
# receipt.signed.acme=*yes | *no
# receipt.delivery.acme=http://www.rocket.com.au/cgi-bin/jsmdirect?
as2receipt
#
message.001=Service successfully loaded
message.002=Service successfully unloaded
message.003=Command is not supported :
#
```

AS2OutboundReceiptService

Service Name: AS2OutboundReceiptService

The AS2OutboundReceiptService is used to send an AS2 receipt.

The AS2OutboundReceiptService supports the following commands:

[SERVICE_LOAD](#)

[SERVICE_GET](#)

[SEND](#)

[GET](#)

[SERVICE_UNLOAD](#)

SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD

SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

SEND

The SEND command is used to send an AS2 receipt.

The following keywords may be supplied on the SEND command or as an entry in a keyword/value list argument.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SEND	RECIPIENT	value	Mandatory. The identity that that will receive the receipt. This value is used in resource file lookups. Value is made up of 2 components separated by a colon, a name component and an optional private keystore password component. Example: ACME:mypassword
	DELIVERY	value	Mandatory. The address for the receipt delivery. This will have been retrieved from the inbound transaction RECEIPTDELIVERY.
	RECEIPTMIC	value	Mandatory. MIC of original transaction.
	RECEIPTMESSAGEID	value	Mandatory. Original Message-Id.
	RECEIPTSUBJECT	value	Mandatory. Original message subject.
	RESPONSE	*DEFAULT	Optional. Default. Send receipt.

	*ERROR	Send an error receipt.
PROCESSED	value	Optional. Default value is "processed".
MODIFIER	value	Optional. Default value is blank or "error" for response *ERROR.
AS2TO	value	Optional. AS2-To.
AS2FROM	value	Optional. AS2-From.
FROM	value	Optional. From.
SUBJECT	value	Optional. Subject.
TEXT	value	Optional. Text
MESSAGEID	value	Optional. Complete transaction message id. The Message-Id is normally generated.
SIGNED	*YES	Optional. If *YES, then a keystore password is required. Access to a private key is not mandatory but it will be required if the receipt is to be signed.
	*NO	Default.
USER	value	Optional. Basic authentication user.
PASSWORD	value	Optional. Basic authentication password.

The RECIPIENT keyword value is used to obtain the signing key file, password and signing certificate using a selection hierarchy of entries from the properties

file.

keystore.{recipient}= pki/filename

keystore = pki/filename

keystore.password.{recipient} = password

keystore.password = password

certificate.{recipient} = pki/filename

certificate = pki/filename

as2-to.{recipient} = companyto

as2-from.{recipient} = companyfrom

as2-from = companyfrom

from.{recipient} = from

from = from

subject.{recipient} = subject

subject = subject

messageid.prefix.{recipient} = company identifier

messageid.prefix. = company identifier

signed.algorithm.{recipient} = *MD5 | *SHA1 | *SHA256 | *SHA384 |
*SHA512

signed.algorithm = *MD5 | *SHA1 | *SHA256 | *SHA384 | *SHA512

Human readable text is obtained from a file using a properties file selection hierarchy.

receipt.message.{recipient} = system/receipt-message.txt

receipt.message = system/receipt-message.txt

receipt.error.{recipient} = system/receipt-error.txt

receipt.error = system/receipt-error.txt

The receipt.message contains the human readable component of the receipt.
The receipt.error contains the human readable component of the error receipt.
The following special merge labels are supported in the receipt.message and

receipt.error files:

%%COMPANYFROM%%

%%COMPANYTO%%

%%DATE%%

%%SUBJECT%%

%%TEXT%%

Sample receipt.message file content:

The message sent to recipient <%%COMPANYTO%%> on %%DATE%% with

Sample receipt.error file content:

The message sent

to recipient <%%COMPANYTO%%> on %%DATE%% with subject "%%SU
following error %%TEXT%%.

GET

The GET command is used to access information from the sent transaction and the HTTP response from the remote AS2 server.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	PROPERTY	value	Mandatory. Property keyword.

The following properties come from the created data source that was sent.

Property	Description	Values
*DATAMESSAGEID	Message-Id.	
value	HTTP protocol property.	

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

Sample AS2OutboundReceiptService resources

```
#
# AS2OutboundReceiptService resource ( Default )
#
messageid.prefix=DEFAULT-AS2
subject=Default subject
from=default@company.com
as2-from="Default Company"
keystore=pki/default-key.p8
keystore.password=password
certificate=pki/default-cert.cer
receipt.message=system/default-receipt-message.txt
#
# Recipient – ACME
#
messageid.prefix.acme=ACME-AS2
as2-to.acme="Acme Corporation"
as2-from.acme="Rocket Inc"
from.acme=receiptperson@rocket.com.au
subject.acme=The Subject
keystore.acme=pki/acme-key.p8
keystore.password.acme=password
certificate.acme=pki/acme-cert.cer
encryption.certificates.acme=pki/acme-cert.cer,pki/default-cert.cer
receipt.message.acme=system/acme-receipt-message.txt
#
message.001=Service successfully loaded
message.002=Service successfully unloaded
message.003=Command is not supported :
#
```

AS2CertificateService

Service Name: AS2CertificateService

The AS2CertificateService is used to create X.509 certificates.

The AS2CertificateService supports the following commands:

SERVICE_LOAD

SERVICE_GET

CREATE

SERVICE_UNLOAD

SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD

SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

CREATE

The CREATE command is used to create X.509 certificates.
Create certificate authority.

Syntax:

Command	Keyword	Value	Developer Command	
CREATE	CERTIFICATE	*AUTHORITY	Mandatory certificate	
	CA_CERTIFICATE_PATH	value	Mandatory file.	
	CA_KEYSTORE_PATH	value	Mandatory keystore.	
	CA_KEYSTORE_PASSWORD	value	Mandatory password.	
	CA_EXPIRY_DATE	value	Mandatory	
	DATEFORMAT	value	Optional: Default. n	
	SERIAL	value	Mandatory	
	ALGORITHM	*DSA		Optional.
		*RSA		Default.
	STRENGTH	value		Optional.
		2048		Default.
	COUNTRY	value		Optional.
	STATE	value		Optional.
	LOCALITY	value		Optional.
	ORGANIZATION	value		Optional.
UNIT	value		Optional.	

	COMMON_NAME	value	Optional.
	EMAIL	value	Optional.
	SIGNING_ALGORITHM	*SHA1RSA	Optional. algorithm.
		*SHA1DSA	
		*SHA256RSA	Default.

Create certificate request.

Syntax:

Command	Keyword	Value	Default
CREATE	CERTIFICATE	*REQUEST	Ma cer
	REQUEST_CERTIFICATE_PATH	value	Ma cer
	REQUEST_KEYSTORE_PATH	value	Ma key
	REQUEST_KEYSTORE_PASSWORD	value	Ma pas
	ALGORITHM	*DSA	Op
		*RSA	Def
	STRENGTH	value	Op
		2048	Def
	COUNTRY	value	Op
	STATE	value	Op
	LOCALITY	value	Op

ORGANIZATION	value	Opt
UNIT	value	Opt
COMMON_NAME	value	Opt
EMAIL	value	Opt
SIGNING_ALGORITHM	*SHA1RSA	Opt alg
	*SHA1DSA	
	*SHA256RSA	Def

Create certificate from certificate request.

Syntax:

Command	Keyword	Value	Develop Comma
CREATE	CERTIFICATE	*CLIENT	Mandate certifica
	CA_CERTIFICATE_PATH	value	Mandate authority
	CA_KEYSTORE_PATH	value	Mandate keystore
	CA_KEYSTORE_PASSWORD	value	Mandate passwor
	REQUEST_CERTIFICATE_PATH	value	Mandate certifica
	CLIENT_CERTIFICATE_PATH	value	Mandate file.
	SERIAL	value	Mandate
	DAYS	value	Mandate

			number
	SIGNING_ALGORITHM	*SHA1RSA	Optional algorithm
		*SHA1DSA	
		*SHA256RSA	Default.

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

AS3FileService

Service Name: AS3FileService

The AS3FileService is used to read and write AS3 transactions and receipts.

The AS3FileService supports the following commands:

SERVICE_LOAD

SERVICE_GET

READ

GET

RECEIVE

CREATE

WRITE

SERVICE_UNLOAD

SERVICE_LOAD

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_LOAD

SERVICE_GET

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
SERVICE_GET	PROPERTY	value	Conditional. Get service property.
	TRACE	*SERVICE	Conditional. Read service trace file.
		*TRANSPORT	Conditional. Read transport trace file.

READ

The READ command reads the AS3 transaction or receipt file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

READ	FILE	value	Mandatory. Filename.
------	------	-------	----------------------

GET

The GET command is used to access information about the read transaction or receipt.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
GET	PROPERTY	value	Mandatory. Property keyword.

Property	Description	Values
*AS3TO	AS3-To	
*AS3FROM	AS3-From	
*AS3VERSION	AS3-Version	
*DATE	Date	
*SUBJECT	Subject	
*MESSAGEID	Message-Id	
*NOTIFICATONTO	Disposition-Notification-To	
*NOTIFICATIONOPTIONS	Disposition-Notification-Options	
*CONTENT	Content-Type	*RECEIPT *XML *X12 *EDIFACT *EDICONSENT content mime type.
value	Protocol property	

The following properties come from a transaction.

Use the GET PROPERTY (*CONTENT) to check if the file contains a transaction.

Property	Description	Values
*MICMD5	Calculated MD5 MIC.	
*MICSHA1	Calculated SHA1 MIC.	
*MICSHA256	Calculated SHA256 MIC.	
*MICSHA384	Calculated SHA384 MIC.	
*MICSHA512	Calculated SHA512 MIC.	
*CONTENTTYPE	Transaction content type	content mime type.

The following properties come from a receipt.

Use the GET PROPERTY (*CONTENT) to check if the file contains a receipt.

Property	Description	Example message values
*RECEIPTMESSAGEID	Message ID of the original transaction. This can be used to match to the original sent transaction.	
*RECEIPTMIC	MIC returned in receipt. This can be compared with the original transaction MIC to check the transaction delivery.	
*RECEIPTMICALG	MIC algorithm.	MD5,

		SHA1
*RECEIPTDISPOSITIONTYPE	Status of the AS3 transaction.	Possible values: processed failed.
*RECEIPTDISPOSITIONMODIFIER	Status of the AS3 transaction.	Possible values: "" error warning

The following properties come from a created transaction or receipt.

Property	Description	Values
*DATAMESSAGEID	Message-Id.	
*DATAMICMD5	CalculatedMD5 MIC.	
*DATAMICSHA1	Calculated SHA1 MIC.	
*DATAMICSHA256	Calculated SHA256 MIC.	
*DATAMICSHA384	Calculated SHA384 MIC.	
*DATAMICSHA512	Calculated SHA512 MIC.	

RECEIVE

The RECEIVE command decodes the loaded transaction.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
RECEIVE	RECEIVER	value	Mandatory. The identity that receives the transaction. This value is used in resource file lookups. Value is made up of 2 components separated by a colon, a name component and an optional private keystore password component. Example: ACME:mypassword
	PARTNER	value	Optional. Authentication partner

The RECEIVER keyword value is used to obtain the decryption key file, password, decryption certificate and verify authentication certificate using a selection hierarchy of entries from the properties file.

keystore.{receiver}= pki/filename

keystore = pki/filename

keystore.password.{receiver} = password

keystore.password = password

certificate.{receiver} = pki/filename

certificate = pki/filename

authentication.certificate.{partner}=pki/filename

CREATE

The CREATE command is used to create an AS3 transaction or receipt.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword
CREATE	OBJECT	*TRANSACTION	Optional. Default. C transaction
	RECIPIENT	value	Mandatory. The identifier that will receive receipt. This value is in resource file look Value is made up of components separated by colon, a name component and an optional private keystore password component. Example: ACME:mypassword
	SUBJECT	value	Optional. Subject.
	RECEIPTTO	value	Optional. Receipt delivery.
		*NONE	No receipt delivery.
	RECEIPTSIGNED	*YES	Optional. Signed Receipt.
		*NO	Default.
	FILE	value	Mandatory. File content EDI content.
	NAME	value	Optional. Name of file. Default is to use the resource name.
	CONTENT	value	Mandatory. Content.
		*XML	application/xml

		*EDIFACT	application/edifact
		*X12	application/x12
	SIGNED	*YES	Optional. Sign conte
		*NO	Default.
	ENCRYPTED	*YES	Optional Encrypt co
		*NO	Default.
	COMPRESSED	*YES	Optional. Compress content.
		*NO	Default.
	MESSAGEID	value	Optional.

The RECIPIENT keyword value is used to obtain the signing key file, password, signing certificate and encryption certificates using a selection hierarchy of entries from the properties file.

keystore.{recipient}= pki/filename

keystore = pki/filename

keystore.password.{recipient} = password

keystore.password = password

certificate.{recipient} = pki/filename

certificate = pki/filename

encryption.certificates.{recipient} = pki/filename, pki/filename, pki/filename

as3-to.{recipient} = companyto

as3-from.{recipient} = companyfrom

as3-from = companyfrom

subject.{recipient} = subject

subject = subject

receipt.to.{recipient}=receipt to

receipt.to=receipt to

receipt.signed.{recipient}=*yes | *no

receipt.signed=*yes | *no

messageid.suffix.{recipient} = company identifier

messageid.suffix. = company identifier

signed.algorithm.{recipient} = *MD5 | *SHA1 | *SHA256 | *SHA384 | *SHA512

signed.algorithm = *MD5 | *SHA1 | *SHA256 | *SHA384 | *SHA512

encryption.algorithm.{recipient} = *DES3 | *AES128 | *AES192 | *AES256

encryption.algorithm = *DES3 | *AES128 | *AES192 | *AES256

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/
CREATE	OBJECT	*RECEIPT	Mandatory. Create rec
	RECIPIENT	value	Mandatory. The identi that receives the transa This value is used in resource file lookups. Value is made up of 2 components separated colon, a name compon and an optional private keystore password component. Example: ACME:mypassword
	RECEIPTMIC	value	Mandatory. Original Message MIC.
	RECEIPTMESSAGEID	value	Mandatory. Original Message Id.
	RECEIPTSUBJECT	value	Mandatory. Original Message Subject.

	PROCESSED	value	Optional. Default is "processed".
	MODIFIER	value	Optional. Default is bl
	AS3TO	value	Optional. AS3-To.
	AS3FROM	value	Optional. AS3-From.
	SUBJECT	value	Optional. Receipt subj
	TEXT	value	Optional. Text.
	MESSAGEID	value	Optional.
	SIGNED	*YES	Optional. Sign receipt.
		*NO	Default.

The RECIPIENT keyword value is used to obtain the signing key file, password and signing certificate using a selection hierarchy of entries from the properties file.

The keystore and certificates are only used when the receipt needs to be signed.

keystore.{recipient}= pki/filename

keystore = pki/filename

keystore.password.{recipient} = password

keystore.password = password

certificate.{recipient} = pki/filename

certificate = pki/filename

as3-to.{recipient} = companyto

as3-from.{recipient} = companyfrom

as3-from = companyfrom

subject.{recipient} = subject

subject = subject

receipt.to.{recipient}=receipt to

receipt.to=receipt to

messageid.suffix.{recipient} = company identifier

messageid.suffix. = company identifier

signed.algorithm.{recipient} = *MD5 | *SHA1 | *SHA256 | *SHA384 |
*SHA512

signed.algorithm = *MD5 | *SHA1 | *SHA256 | *SHA384 | *SHA512

Human readable text is obtained from a file using a properties file selection hierarchy.

receipt.message.{recipient} = system/receipt-message.txt

receipt.message = system/receipt-message.txt

The receipt.message contains the human readable component of the receipt.

The following special merge labels are supported in the receipt.message file:

%%COMPANYFROM%%

%%COMPANYTO%%

%%DATE%%

%%SUBJECT%%

%%TEXT%%

Sample receipt.message file content:

The message sent to recipient <%%COMPANYTO%%> on %%DATE%% wi

WRITE

The WRITE command saves the created transaction or receipt to the specified file.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

WRITE	FILE	value	Mandatory. Filename.
-------	------	-------	----------------------

SERVICE_UNLOAD

The SERVICE_UNLOAD command unloads the service.

Syntax:

Command	Keyword	Value	Developer notes for Command/Keyword/Value
----------------	----------------	--------------	--

SERVICE_UNLOAD			
----------------	--	--	--

Sample AS3FileService resources

```
#
# AS3FileService resource ( Default )
#
# Default
#
messageid.suffix=DEFAULT
subject=Default Subject
as3-from=Default Company
keystore=pki/default-key.p8
keystore.password=password
certificate=pki/default-cert.cer
receipt.subject=Default Receipt Subject
# receipt.signed=*yes | *no
receipt.to=ftp://mycompany/default
# signed.algorithm=*sha1 | *md5
# encryption.algorithm=*des3
#
# Recipient - ACME
#
messageid.suffix.acme=ACME
as3-to.acme="Acme Inc."
as3-from.acme="My Company"
subject.acme=Subject
keystore.acme=pki/acme-key.p8
keystore.password.acme=password
certificate.acme=pki/acme-cert.cer
encryption.certificates.acme=pki/acme-cert.cer
receipt.subject.acme=Receipt Subject
# receipt.signed.acme=*yes | *no
# receipt.to.acme=*none | {address}
receipt.to.acme=ftp://mycompany/inbox
# signed.algorithm.acme=*sha1 | *md5
# encryption.algorithm.acme=*des3
#
```