The Spring.NET Framework

Reference Documentation

Authors

Mark Pollack , Rick Evans , Aleksandar Seovic , Bruno Baia , Federico Spinazzi , Rob Harrop , Griffin Caprio , Ruben Bartelink , Choy Rim , The Spring Java Team

Version 1.2.0

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Last Updated Novermber 10, 2008 (Latest documentation)

1. Preface

2. Introduction

- 2.1. Overview
- 2.2. Background
- 2.3. Modules
- 2.4. Usage Scenarios
- 2.5. Quickstart applications
- 2.6. License Information
- 2.7. Support

3. Background information 3.1. Inversion of Control

4. Migrating from 1.1 M2

4.1. Introduction

4.2. Important Changes

4.2.1. Namespaces

4.2.2. Core

4.2.3. Web

4.2.4. Data

I. Core Technologies

				400	
5	IhΔ	loC	con	ıtaır	ıΔr
J.	HIC		COL	пан	ıcı

- 5.1. Introduction
- 5.2. Basics containers and objects
 - 5.2.1. The container
 - 5.2.1.1. Configuration metadata
 - 5.2.2. Instantiating a container
 - 5.2.2.1. Composing XML-based configuration metadata
 - 5.2.3. The Objects
 - 5.2.3.1. Naming objects
 - 5.2.4. Object creation
 - 5.2.4.1. Object creation via constructor invocation
 - 5.2.4.2. Object creation via a static factory method
 - 5.2.4.3. Object creation via an instance factory method
 - 5.2.5. Object creation of generic types
 - 5.2.5.1. Object creation of generic types via constructor invocation
 - 5.2.5.2. Object creation of generic types via static factory method
 - 5.2.5.3. Object creation of generic types via instance factory method
 - 5.2.6. Using the container
- 5.3. Dependencies
 - 5.3.1. Injecting dependencies
 - 5.3.1.1. Constructor Injection
 - 5.3.1.2. Setter Injection
 - 5.3.1.3. Some examples
 - 5.3.2. Dependencies and configuration in detail
 - 5.3.2.1. Straight values (primitives, strings, etc.)
 - 5.3.2.2. Referring to collaborating objects
 - 5.3.2.3. Inline objects
 - 5.3.2.4. Setting collection values
 - 5.3.2.5. Setting generic collection values
 - 5.3.2.6. Setting null values
 - 5.3.2.7. Setting indexer properties
 - 5.3.2.8. Value and ref shortcut forms
 - 5.3.2.9. Compound property names and Spring expression references
 - 5.3.3. Declarative Event Listener Registration
 - 5.3.3.1. Declarative event handlers
 - 5.3.3.2. Configuring a method to be invoked when an event is fired
 - 5.3.3.3. Registering a collection of handler methods based on a regular expression
 - 5.3.3.4. Registering a handler method against an event name that contains a regular expression
 - 5.3.4. Using depends-on
 - 5.3.5. Lazily-instantiated objects
 - 5.3.6. Autowiring collaborators

5.3.7. Checking for dependencies
5.3.8. Method Injection
5.3.8.1. Lookup Method Injection
5.3.8.2. Arbitrary method replacement
5.3.9. Setting a reference using the members of other objects and classes.
5.3.9.1. Setting a reference to the value of property.
5.3.9.2. Setting a reference to the value of field.
5.3.9.3. Setting a property or constructor argument to the return value of a method invocation.
5.3.10. Provided IFactoryObject implementations
5.3.10.1. Common logging
5.4. Object Scopes
5.4.1. The singleton scope
5.4.2. The prototype scope
5.4.3. Singleton objecgts with prototype-object dependencies
5.4.4. The other scopes
5.5. Type conversion
5.5.1. Type Conversion for Enumerations
5.5.2. Built-in TypeConverters
5.5.3. Custom Type Conversion
5.5.3.1. Using CustomConverterConfigurer
5.6. Customizing the nature of an object
5.6.1. Lifecycle interfaces
5.6.1.1. IInitializingObject / init-method
5.6.1.2. IDisposable / destroy-method
5.6.2. Knowing who you are
5.6.2.1. IObjectFactoryAware
5.6.2.2. IObjectNameAware
5.7. Object definition inheritance
5.8. Interacting with the container
5.8.1. Obtaining an IFactoryObject, not its product
5.9. Container extension points
5.9.1. Customizing objects with IObjectPostProcessors
5.9.1.1. Example: Hello World, IObjectPostProcessor-style
5.9.1.2. Example: the RequiredAttributeObjectPostProcessor
5.9.2. Customizing configuration metadata with ObjectFactoryPostProcessors
5.9.2.1. Example: The PropertyPlaceholderConfigurer
5.9.2.2. Example: The PropertyOverrideConfigurer
5.9.2.3. IVariableSource
5.9.3. Customizing instantiation logic using IFactoryObjects
5.9.3.1. IConfigurableFactoryObject 5.10. The IApplicationContext
5.10. The IApplicationContext
5.10.1. IObjectFactory or IApplicationContext?

5.11. Configuration of IApplicationContext
5.11.1. Registering custom parsers
5.11.2. Registering custom resource handlers
5.11.3. Registering Type Aliases
5.11.4. Registering Type Converters
5.12. Added functionality of the IApplicationContext
5.12.1. Context Hierarchies
5.12.2. Using IMessageSource
5.12.3. Using resources within Spring.NET
5.12.4. Loosely coupled events
5.12.5. Event notification from IApplicationContext
5.13. Customized behavior in the ApplicationContext
5.13.1. The IApplicationContextAware marker interface
5.13.2. The IObjectPostProcessor
5.13.3. The IObjectFactoryPostProcessor
5.13.4. The PropertyPlaceholderConfigurer
5.14. Configuration of ApplicationContext without using XML
5.15. Service Locator access
5.16. Stereotype attributes

6. The IObjectWrapper and Type conversion

- 6.1. Introduction
- 6.2. Manipulating objects using the IObjectWrapper
 - 6.2.1. Setting and getting basic and nested properties
 - 6.2.2. Other features worth mentioning
- 6.3. Type conversion
 - 6.3.1. Type Conversion for Enumerations
- 6.4. Built-in TypeConverters
 - 6.4.1. Custom type converters

7. Resources

- 7.1. Introduction
- 7.2. The IResource interface
- 7.3. Built-in IResource implementations
 - 7.3.1. Registering custom IResource implementations
- 7.4. The IResourceLoader
- 7.5. The IResourceLoaderAware interface
- 7.6. Application contexts and IResource paths

8. Threading and Concurrency Support

- 8.1. Introduction
- 8.2. Thread Local Storage
- 8.3. Synchronization Primitives
 - 8.3.1. ISync
 - 8.3.2. SyncHolder
 - 8.3.3. Latch
 - 8.3.4. Semaphore

- 9. Object Pooling
 - 9.1. Introduction
 - 9.2. Interfaces and Implementations

10. Spring.NET miscellanea

10.1. Introduction

10.2. PathMatcher

10.2.1. General rules

10.2.2. Matching filenames

10.2.3. Matching subdirectories

10.2.4. Case does matter, slashes don't

```
11. Expression Evaluation
      11.1. Introduction
      11.2. Evaluating Expressions
      11.3. Language Reference
            11.3.1. Literal expressions
           11.3.2. Properties, Arrays, Lists, Dictionaries, Indexers
                  11.3.2.1. Defining Arrays, Lists and Dictionaries Inline
            11.3.3. Methods
            11.3.4. Operators
                  11.3.4.1. Relational operators
                  11.3.4.2. Logical operators
                 11.3.4.3. Mathematical operators
           11.3.5. Assignment
            11.3.6. Expression lists
            11.3.7. Types
            11.3.8. Type Registration
            11.3.9. Constructors
            11.3.10. Variables
                  11.3.10.1. The '#this' and '#root' variables
            11.3.11. Ternary Operator (If-Then-Else)
            11.3.12. List Projection and Selection
            11.3.13. Collection Processors and Aggregators
                 11.3.13.1. Count Aggregator
                 11.3.13.2. Sum Aggregator
                  11.3.13.3. Average Aggregator
                 11.3.13.4. Minimum Aggregator
                 11.3.13.5. Maximum Aggregator
                 11.3.13.6. Non-null Processor
                 11.3.13.7. Distinct Processor
                 11.3.13.8. Sort Processor
                 11.3.13.9. Type Conversion Processor
                  11.3.13.10. Reverse Processor
                  11.3.13.11. OrderBy Processor
                 11.3.13.12. User Defined Collection Processor
            11.3.14. Spring Object References
            11.3.15. Lambda Expressions
            11.3.16. Delegate Expressions
            11.3.17. Null Context
```

11.4. Classes used in the examples

10 \	/- II -	1 - 41	E		а.
12. V	/allc	lation	⊢ran	iewoi	K.

- 12.1. Introduction
- 12.2. Example Usage
- 12.3. Validator Groups
- 12.4. Validators
 - 12.4.1. Condition Validator
 - 12.4.2. Required Validator
 - 12.4.3. Regular Expression Validator
 - 12.4.4. Generic Validator
 - 12.4.5. Conditional Validator Execution
- 12.5. Validator Actions
 - 12.5.1. Error Message Action
 - 12.5.2. Generic Actions
- 12.6. Validator References
- 12.7. Progammatic usage
- 12.8. Usage tips within ASP.NET
 - 12.8.1. Rendering Validation Errors
 - 12.8.1.1. Configuring which Error Renderer to use.

13. Aspect Oriented Programming with Spring.NET
13.1. Introduction
13.1.1. AOP concepts
13.1.2. Spring.NET AOP capabilities
13.1.3. AOP Proxies in Spring.NET
13.2. Pointcut API in Spring.NET
<u>13.2.1. Concepts</u>
13.2.2. Operations on pointcuts
13.2.3. Convenience pointcut implementations
13.2.3.1. Static pointcuts
13.2.3.2. Dynamic Pointcuts
13.2.4. Custom pointcuts
13.3. Advice API in Spring.NET
13.3.1. Advice Lifecycle
13.3.2. Advice types
13.3.2.1. Interception Around Advice
13.3.2.2. Before advice
13.3.2.3. Throws advice
13.3.2.4. After Returning advice
13.3.2.5. Advice Ordering
13.3.2.6. Introduction advice
13.4. Advisor API in Spring.NET
13.5. Using the ProxyFactoryObject to create AOP proxies
<u>13.5.1. Basics</u>
13.5.2. ProxyFactoryObject Properties
13.5.3. Proxying Interfaces
13.5.4. Proxying Classes
13.5.5. Concise proxy definitions
13.6. Proxying mechanisms
13.6.1. InheritanceBasedAopConfigurer
13.7. Creating AOP Proxies Programatically with the ProxyFactory
13.8. Manipulating Advised Objects
13.9. Using the "autoproxy" facility
13.9.1. Autoproxy object definitions
13.9.1.1. ObjectNameAutoProxyCreator
13.9.1.2. DefaultAdvisorAutoProxyCreator
13.9.1.4. TypoNamoAutoProxyCreator
13.9.1.4. TypeNameAutoProxyCreator
13.9.1.5. AttributeAutoProxyCreator 13.9.1.6. AbstractFilteringAutoProxyCreator
13.9.1.7. AbstractAutoProxyCreator
13.9.2. Using attribute-driven auto-proxying
13.3.2. Using aunuale-unven aulu-proxymig

- 13.10. Using AOP Namespace
- 13.11. Using TargetSources
 - 13.11.1. Hot swappable target sources
 - 13.11.2. Pooling target sources
 - 13.11.3. Prototype target sources
 - 13.11.4. ThreadLocal target sources
- 13.12. Defining new Advice types
- 13.13. Further reading and resources

14. Aspect Library

14.1. Introduction

14.2. Caching

14.3. Exception Handling

14.3.1. Language Reference

14.4. Logging

14.5. Retry

14.5.1. Language Reference

14.6. Transactions

14.7. Parameter Validation

15. Common Logging 15.1. Introduction

16. Testing

16.1. Introduction

16.2. Unit testing

16.3. Integration testing

16.3.1. Context management and caching

16.3.2. Dependency Injection of test fixtures

16.3.2.1. Field level injection

16.3.3. Transaction management

16.3.4. Convenience variables

16.3.5.

16.4. Further Resources

II. Middle Tier Data Access

17. Transaction management				
17.1. Introduction				
17.2. Motivations				
17.3. Key Abstractions				
17.4. Resource synchroniza	ation with trans	<u>actions</u>		
<u>17.4.1. High-level ap</u> ı	<u>oroach</u>			
17.4.2. Low-level app	<u>oroach</u>			
17.5. Declarative transaction	<u>ın managemen</u>	ţ		
17.5.1. Understandin	g Spring's decl	<u>arative transaction in </u>	<u>nplementation</u>	
17.5.2. A First Examp	<u>ole</u>			
17.5.3. Declarative tra	<u>ansactions usir</u>	ng the transaction na	<u>mespace</u>	
17.5.4. Transaction a	ttribute settings	<u> </u>		
17.5.5. Declarative Tr	<u>ransactions usi</u>	ng AutoProxy		
17.5.5.1.		transactional	proxies	with
<u>ObjectNameAu</u>	•			
<u>17.5.5.2.</u>		transactional	proxies	with
<u>DefaultAdvisor</u>				
17.5.6. Declarative Ti		ng TransactionProxy	<u>FactoryObject</u>	
17.5.7. Concise proxy				
17.5.8. Declarative Ti			ect .	
17.6. Programmatic transac				
17.6.1. Using the Tra	•			
17.6.1.1. Speci		•		
17.6.2. Using the Pla				
17.7. Choosing between pr	•		<u>:tion_management</u>	
17.8 Transaction lifecycle a	and status infor	mation		

18. DAO support

- 18.1. Introduction
- 18.2. Consistent exception hierarchy
- 18.3. Consistent abstract classes for DAO support

19. DbProvider

- 19.1. Introduction
- 19.2. IDbProvider and DbProviderFactory
- 19.3. XML based configuration
- 19.4. Connection String management
- 19.5. Additional IDbProvider implementations
 - 19.5.1. UserCredentialsDbProvider
 - 19.5.2. MultiDelegatingDbProvider

20. Data access using ADO.NET 20.1. Introduction **Motivations** 20.3. Provider Abstraction 20.3.1. Creating an instance of IDbProvider 20.4. Namespaces 20.5. Approaches to Data Access 20.6. Introduction to AdoTemplate 20.6.1. Execute Callback 20.6.2. Execute Callback in .NET 2.0 20.6.3. Execute Callback in .NET 1.1 20.6.4. Quick Guide to AdoTemplate Methods 20.6.5. Quick Guide to AdoTemplate Properties 20.7. Transaction Management 20.8. Exception Translation 20.9. Parameter Management 20.9.1. IDbParametersBuilder 20.9.2. IDbParameters 20.10. Custom IDataReader implementations 20.11. Basic data access operations 20.11.1. ExecuteNonOuerv 20.11.2. ExecuteScalar 20.12. Queries and Lightweight Object Mapping 20.12.1. ResultSetExtractor 20.12.2. RowCallback 20.12.3. RowMapper 20.12.4. Query for a single object 20.12.5. Query using a CommandCreator 20.13. DataTable and DataSet 20.13.1. DataTables 20.13.2. DataSets 20.14. TableAdapters and participation in transactional context 20.15. Database operations as Objects 20.15.1. AdoQuery 20.15.2. MappingAdoQuery 20.15.3. AdoNonQuery 20.15.4. Stored Procedure

21.	Object	Relational	Mapping	(ORM)	data	access

21.1. Introduction

21.2. NHibernate

- 21.2.1. Resource management
- 21.2.2. Transaction Management
- 21.2.3. SessionFactory set up in a Spring container
- 21.2.4. The HibernateTemplate
- 21.2.5. Implementing Spring-based DAOs without callbacks
- 21.2.6. Implementing DAOs based on plain Hibernate 1.2/2.0 API

21.2.6.1. Exception Translation

- 21.2.7. Programmatic transaction demarcation
- 21.2.8. Declarative transaction demarcation
- 21.2.9. Transaction management strategies
- 21.2.10. Web Session Management
- 21.2.11. Session Scope

III. The Web

22. Spring.NET Web Framework
22.1. Introduction
22.2. Background
22.3. Automatic context loading and hierarchical contexts
22.3.1. Configuration
22.3.1.1. Configuration for IIS7
22.3.2. Context Hierarchy
22.4. Dependency Injection for ASP.NET Pages
22.4.1. Injecting Dependencies into Controls
22.4.2. Injecting dependencies into custom HTTP modules
22.4.3. Injecting dependencies into HTTP handlers and handler factories
22.4.4. Injecting dependencies into custom providers
22.4.5. Customizing control dependency injection
22.5. Web object scopes
22.6. Master Pages in ASP.NET 1.1
22.6.1. Linking child pages to their master
22.7. Bidirectional Data Binding and Model Management
22.7.1. Data Binding Under the Hood
22.7.1.1. Binding Direction
22.7.1.2. Formatters
22.7.1.3. Type Conversion
22.7.1.4. Data Binding Events
22.7.1.5. Rendering Binding Errors
22.7.1.6. HttpReguestListBindingContainer
22.7.2. Using DataBindingPanel
22.7.3. Customizing Model Persistence
22.8. Localization and Message Sources
22.8.1. Automatic Localization Using Localizers ("Push" Localization)
22.8.2. Global Message Sources
22.8.3. Working with Localizers
22.8.4. Applying Resources Manually ("Pull" Localization)
22.8.5. Localizing Images within a Web Application
22.8.6. User Culture Management
22.8.6.1. DefaultWebCultureResolver
22.8.6.2. RequestCultureResolver
22.8.6.3. SessionCultureResolver
22.8.6.4. CookieCultureResolver
22.8.7. Changing Cultures
22.9. Result Mapping
22.9.1. Registering user defined transfer modes
22.10. Client-Side Scripting
22.10.1. Registering Scripts within the head HTML section

22.10.2. Adding CSS Definitions to the head Section

22.10.3. Well-Known Directories

22.11. Spring User Controls

22.11.1. Validation Controls

22.11.2. Databinding Controls

22.11.3. Calendar Control

22.11.4. Panel Control

23. ASP.NET AJAX

23.1. Introduction

23.2. Web Services

23.2.1. Exposing Web Services

23.2.2. Calling Web Services by using JavaScript

IV. Services

24. Introduction to Spring Services 24.1. Introduction

25. .NET Remoting

- 25.1. Introduction
- 25.2. Publishing SAOs on the Server
 - 25.2.1. SAO Singleton
 - 25.2.2. SAO SingleCall
 - 25.2.3. IIS Application Configuration
- 25.3. Accessing a SAO on the Client
- 25.4. CAO best practices
- 25.5. Registering a CAO object on the Server
 - 25.5.1. Applying AOP advice to exported CAO objects
- 25.6. Accessing a CAO on the Client
 - 25.6.1. Applying AOP advice to client side CAO objects.
- 25.7. XML Schema for configuration
- 25.8. Additional Resources

26. .NET Enterprise Services

- 26.1. Introduction
- 26.2. Serviced Components
- 26.3. Server Side
- 26.4. Client Side

27. Web Services

- 27.1. Introduction
- 27.2. Server-side
 - 27.2.1. Removing the need for .asmx files
 - 27.2.2. Injecting dependencies into web services
 - 27.2.3. Exposing PONOs as Web Services
 - 27.2.4. Exporting an AOP Proxy as a Web Service
- 27.3. Client-side
 - 27.3.1. Using VS.NET generated proxy
 - 27.3.2. Generating proxies dynamically
 - 27.3.3. Configuring the proxy instance

28. Windows Communication Foundation (WCF)

- 28.1. Introduction
- 28.2. Configuring WCF services via Dependency Injection 28.2.1. Dependency Injection
- 28.3. Apply AOP advice to WCF services
- 28.4. Creating client side proxies declaratively
- 28.5. Exporting PONOs as WCF Services

V. Integration

<u> 29.</u>	<u>Message</u>	Oriented	<u> Middleware -</u>	<u>Apache</u>	<u>ActiveMQ</u>
	00.4	and the same			

- 29.1. Introduction
 - 29.1.1. Separation of Concerns
 - 29.1.2. Interoperability and provider portability
 - 29.1.3. The role of Messaging API in a 'WCF world'
- 29.2. Using Spring Messaging
 - 29.2.1. Messaging Template overview
 - 29.2.2. Connections
 - 29.2.3. Caching Messaging Resources
 - 29.2.3.1. SingleConnectionFactory
 - 29.2.3.2. CachingConnectionFactory
 - 29.2.4. Destination Management
 - 29.2.5. Message Listener Containers
 - 29.2.6. Transaction Management
- 29.3. Sending a Message
 - 29.3.1. Using MessageConverters
- 29.4. Session and Producer Callback
- 29.5. Receiving a message
 - 29.5.1. Synchronous Reception
 - 29.5.2. Asynchronous Reception
 - 29.5.3. The ISessionAwareMessageListener interface
 - 29.5.4. MessageListenerAdapater
 - 29.5.5. Processing messages within a messaging transaction
 - 29.5.6. Messaging Namespace support

30. Message Oriented Middleware - MSMQ
30.1. Introduction
30.2. A quick tour for the impatient
30.3. Using Spring MSMQ
30.3.1. MessageQueueTemplate
30.3.2. MessageQueueFactoryObject
30.3.3. MessageQueue and IMessageConverter resource management
30.3.4. Message Listener Containers
30.3.4.1. NonTransactionalMessageListenerContainer
30.3.4.2. TransactionalMessageListenerContainer
30.3.4.3. DistributedTxMessageListenerContainer
30.4. MessageConverters
30.4.1. Using MessageConverters
30.5. Interface based message processing
<u>30.5.1.</u>

30.5.1.1. MessageListenerAdapater

30.6. Comparison with using WCF

31. Scheduling and Thread Pooling

31.1. Introduction

31.2. Using the Quartz.NET Scheduler

31.2.1. Using the JobDetailObject

31.2.2. Using the MethodInvokingJobDetailFactoryObject

31.2.3. Wiring up jobs using triggers and the SchedulerFactoryObject

VI. VS.NET Integration

32. Visual Studio.NET Integration

32.1. XML Editing and Validation

32.2. Versions of XML Schema

32.3. Integrated API help

VII. Quickstart applications

33. IoC Quickstarts
33.1. Introduction
33.2. Movie Finder
33.2.1. Getting Started - Movie Finder
33.2.2. First Object Definition
33.2.3. Setter Injection
33.2.4. Constructor Injection
33.2.5. Summary
33.2.6. Logging
33.3. ApplicationContext and IMessageSource
33.3.1. Introduction
33.4. ApplicationContext and IEventRegistry
33.4.1. Introduction
33.5. Pooling example
33.5.1. Implementing Spring.Pool.IPoolableObjectFactory
33.5.2. Being smart using pooled objects

33.5.3. Using the executor to do a parallel grep

33.6. AOP

34. AOP Guide

34.1. Introduction

34.2. The basics

34.2.1. Applying advice

34.2.2. Using Pointcuts - the basics

34.3. Going deeper

34.3.1. Other types of Advice

34.3.1.1. Before advice

34.3.1.2. After advice

34.3.1.3. Throws advice

34.3.1.4. Introductions (mixins)

34.3.1.5. Layering advice

34.3.1.6. Configuring advice

34.3.2. Using Attributes to define Pointcuts

34.4. The Spring.NET AOP Cookbook

34.4.1. Caching

34.4.2. Performance Monitoring

34.4.3. Retry Rules

34.5. Spring.NET AOP Best Practices

35. Portable Service Abstraction Quick Start

- 35.1. Introduction
- 35.2. .NET Remoting Example
- 35.3. Implementation
- 35.4. Running the application
- 35.5. Remoting Schema
- 35.6. .NET Enterprise Services Example
- 35.7. Web Services Example
- 35.8. Additional Resources

36. Web Quickstarts 36.1. Introduction

37. SpringAir - Reference Application

- 37.1. Introduction
- 37.2. Getting Started
- 37.3. Container configuration
- 37.4. Bi-directional data binding
- 37.5. Declarative Validation
- 37.6. Internationalization
- 37.7. Web Services

38. Data Access QuickStart

38.1. Introduction

38.1.1. Database configuration

38.1.1.1. AdoTemplate Configuration

38.1.2. CommandCallback

39. Transactions QuickStart

39.1. Introduction

39.2. Application Overview

39.2.1. Interfaces

39.3. Implementation

39.4. Configuration

39.4.1. Rollback Rules

39.5. Adding additional Aspects

40. NHibernate QuickStart 40.1. Introduction

41. Quartz QuickStart

- 41.1. Introduction
- 41.2. Application Overview
- 41.3. Standard job scheduling
- 41.4. Scheduling arbitrary methods as jobs

42. NMS QuickStart

- 42.1. Introduction
- 42.2. Message Destinations
- 42.3. Gateways
- 42.4. Message Data
- 42.5. Message Handlers
- 42.6. Message Converters
- 42.7. Messaging Infrastructure
- 42.8. Running the application

43. MSMQ QuickStart

- 43.1. Introduction
- 43.2. Message Destinations
- 43.3. Gateways
- 43.4. Message Data
- 43.5. Message Handlers
- 43.6. MessageConverters
- 43.7. Messaging Infrastructure
- 43.8. Running the application

44. WCF QuickStart

44.1. Introduction

44.2. The server side

44.2.1. WCF Dependency Injection and AOP in self-hosted application 44.2.2. WCF Dependency Injection and AOP in IIS web application

44.3. Client access

VIII. Spring.NET for Java developers

45. S	pring.	.NET	for	Java	Deve	lopers

- 45.1. Introduction
- 45.2. Beans to Objects
- 45.3. PropertyEditors to TypeConverters
- 45.4. ResourceBundle-ResourceManager
- 45.5. Exceptions
- 45.6. Application Configuration
- 45.7. AOP Framework

45.7.1. Cannot specify target name at the end of interceptorNames for ProxyFactoryObject

A. XML Schema-based configuration

- A.1. Introduction
- A.2. XML Schema-based configuration
 - A.2.1. Referencing the schemas
 - A.2.2. The tx (transaction) schema
 - A.2.3. The aop schema
 - A.2.4. The db schema
 - A.2.5. The remoting schema
 - A.2.6. The nms messaging schema
 - A.2.7. The validation schema
 - A.2.8. The objects schema
- A.3. Setting up your IDE
- B. Extensible XML authoring
 - **B.1.** Introduction
 - B.2. Authoring the schema
 - B.3. Coding a INamespaceParser
 - B.4. Coding an IObjectDefinitionParser
 - B.5. Registering the handler and the schema
 - B.5.1. NamespaceParsersSectionHandler
 - B.6. Using a custom extension in your Spring XML configuration
 - B.7. Further Resources
- C. Spring.NET's spring-objects.xsd

Spring Framework



Chapter 1. Preface

Developing software applications is hard enough even with good tools and technologies. Spring provides a light-weight solution for building enterprise-ready applications. Spring provides a consistent and transparent means to configure your application and integrate AOP into your software. Highlights of Spring's functionality are providing declarative transaction management for your middle tier as well as a full-featured ASP.NET framework.

Spring could potentially be a one-stop-shop for many areas of enterprise application development; however, Spring is modular, allowing you to use just those parts of it that you need, without having to bring in the rest. You can use just the IoC container to configure your application and use traditional ADO.NET based data access code, but you could also choose to use just the Hibernate integration code or the ADO.NET abstraction layer. Spring has been (and continues to be) designed to be non-intrusive, meaning dependencies on the framework itself are generally none (or absolutely minimal, depending on the area of use).

This document provides a reference guide to Spring's features. Since this document is still to be considered very much work-in-progress, if you have any requests or comments, please post them on the user mailing list or on the support forums at forum.springframework.net.

Before we go on, a few words of gratitude are due to Christian Bauer (of the Hibernate team), who prepared and adapted the DocBook-XSL software in order to be able to create Hibernate's reference guide, thus also allowing us to create this one. Also thanks to Russell Healy for doing an extensive and valuable review of some of the material.

Chapter 2. Introduction

2.1. Overview

Spring.NET is an application framework that provides comprehensive infrastructural support for developing enterprise .NET applications. It allows you to remove incidental complexity when using the base class libraries makes best practices, such as test driven development, easy practices. Spring.NET is created, supported and sustained by <u>SpringSource</u>.

The design of Spring.NET is based on the Java version of the Spring Framework, which has shown real-world benefits and is used in thousands of enterprise applications world wide. Spring .NET is not a quick port from the Java version, but rather a 'spiritual port' based on following proven architectural and design patterns in that are not tied to a particular platform. The breadth of functionality in Spring .NET spans application tiers which allows you to treat it as a 'one stop shop' but that is not required. Spring .NET is not an all-or-nothing solution. You can use the functionality in its modules independently. These modules are described below.

Enterprise applications typically are composed of a number of a variety of physical tiers and within each tier functionality is often split into functional layers. The business service layer for example typically uses a objects in the data access layer to fulfill a use-case. No matter how your application is architected, at the end of the day there are a variety of objects that collaborate with one another to form the application proper. The objects in an application can thus be said to have dependencies between themselves.

The .NET platform provides a wealth of functionality for architecting and building applications, ranging all the way from the very basic building blocks of primitive types and classes (and the means to define new classes), to rich full-featured application servers and web frameworks. One area that is decidedly conspicuous by its absence is any means of taking the basic building blocks and composing them into a coherent whole; this area has typically been left to the purvey of the architects and developers tasked with building an application (or applications). Now to be fair, there are a number of design patterns devoted to the business of composing the various classes and object instances that makeup an all-singing, all-dancing application. Design patterns such as Factory, Abstract Factory, Builder, Decorator, and Service Locator (to name but a few) have widespread recognition and acceptance within the software development industry (presumably that is why these patterns have been formalized as patterns

in the first place). This is all very well, but these patterns are just that: best practices given a name, typically together with a description of what the pattern does, where the pattern is typically best applied, the problems that the application of the pattern addresses, and so forth. Notice that the last paragraph used the phrase "... a description of what the pattern does..."; pattern books and wikis are typically listings of such formalized best practice that you can certainly take away, mull over, and then implement yourself in your application.

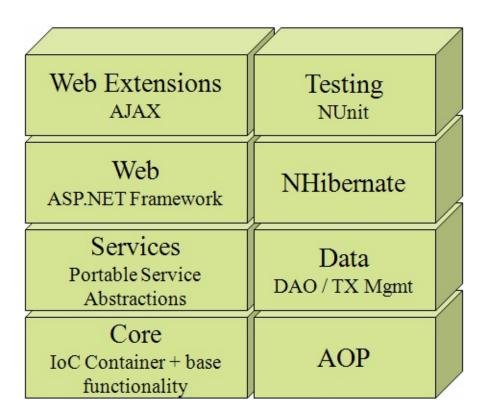
The Spring Framework takes best practices that have been proven over the years in numerous applications and formalized as design patterns, and actually codifies these patterns as first class objects that you as an architect and developer can take away and integrate into your own application(s). This is a Very Good Thing Indeed as attested to by the numerous organizations and institutions that have used the Spring Framework to engineer robust, maintainable applications. For example, the IoC component of the Spring Framework addresses the enterprise concern of taking the classes, objects, and services that are to compose an application, by providing a formalized means of composing these various disparate components into a fully working application ready for use

2.2. Background

In early 2004, Martin Fowler asked the readers of his site: when talking about Inversion of Control: "the question is, what aspect of control are [they] inverting?". Fowler then suggested renaming the principle (or at least giving it a more self-explanatory name), and started to use the term Dependency Injection. His article then continued to explain the ideas underpinning the Inversion of Control (IoC) and Dependency Injection (DI) principle. If you need a decent insight IoC please do refer the into and DI, to article http://martinfowler.com/articles/injection.html.

2.3. Modules

The Spring Framework contains a lot of features, which are well-organized into modules shown in the diagram below. The diagram below shows the various core modules of Spring.NET.



Click on the module name for more information.

<u>Spring.Core</u> is the most fundamental part of the framework allowing you to configure your application using Dependency Injection. Other supporting functionality, listed below, is located in Spring.Core

Spring.Aop - Use this module to perform Aspect-Oriented Programming (AOP). AOP centralizes common functionality that can then be declaratively applied across your application in a targeted manner. Spring's <u>aspect library</u> provides predefined easy to use aspects for transactions, logging, performance monitoring, caching, method retry, and exception handling.

<u>Spring.Data</u> - Use this module to achieve greater efficiency and consistency in writing data access functionality in ADO.NET and to perform declarative transaction management.

<u>Spring.Data.NHibernate</u> - Use this module to integrate NHibernate with Spring's declarative transaction management functionality allowing easy mixing of ADO.NET and NHibernate operations within the same transaction. NHibernate 1.0 users will benefit from ease of use APIs to perform data access operations.

Spring.Web - Use this module to raise the level of abstraction when writing ASP.NET web applications allowing you to effectively address common painpoints in ASP.NET such as data binding, validation, and ASP.NET page/control/module/provider configuration.

<u>Spring.Web.Extensions</u> - Use this module to raise the level of abstraction when writing ASP.NET web applications allowing you to effectively address common pain-points in ASP.NET such as data binding, validation, and ASP.NET page/control/module/provider configuration.

<u>Spring.Services</u> - Use this module to adapt plain .NET objects so they can be used with a specific distributed communication technology, such as .NET Remoting, Enterprise Services, and ASMX Web Services. These services can be configured via dependency injection and 'decorated' by applying AOP.

<u>Spring.NUnit</u> - Use this module to perform integration testing with NUnit.

The Spring.Core module also includes the following additional features

- <u>Expression Language</u> provides efficient querying and manipulation of an object graphs at runtime.
- <u>Validation Framework</u> a robust UI agnostic framework for creating complex validation rules for business objects either programatically or declaratively.
- Data binding Framework a UI agnostic framework for performing data binding.
- Dynamic Reflection provides a high performance reflection API
- Threading provides additional concurrency abstractions such as Latch, Semaphore and Thread Local Storage.
- Resource abstraction provides a common interface to treat the InputStream from a file and from a URL in a polymorphic and protocol-independent manner.

2.4. Usage Scenarios

With the building blocks described above you can use Spring in all sorts of scenarios, from simple stand alone console applications to fully-fledged enterprise applications using Spring's transaction management functionality and web framework integration.

It is important to note that the Spring Framework *does not* force you to use everything within it; it is not an *all-or-nothing* solution. Existing front-ends built using standard ASP.NET can be integrated perfectly well with a Spring-based middle-tier, allowing you to use the transaction and/or data access features that Spring offers. The only things you need to do is wire up your business logic using Spring's IoC container and integrate it into your web layer using WebApplicationContext to locate middle tier services and/or configure your standard ASP.NET pages with depdenency injection.

While the Spring framework does not force any particular application architecture it encourages the use of a well layered application architecture with distinct tiers for the presentation, service, data access, and database.

2.5. Quickstart applications

There are several sample applications that showcase individual features. If you are already familiar with the concepts of dependency injection, AOP, or have experience using the Java version of the Spring framework you may find jumping into the examples a better way to bootstrap the learning processing process. The following quickstart applications are available and can be found in the examples directory in the distribution. Click on the links for additional information.

- <u>Movie Finder</u> A simple demonstration of Dependency Injection (DI) techniques using Spring's Inversion of Control (IoC) container.
- Application Context Demonstrates IoC container features such as localization, accessing of ResourceSet objects, and applying resources to object properties.
- Aspect Oriented Programming Demonstrates use of the AOP framework to add additional behavior to your existing objects. Examples of programmatic and declarative AOP configuration are shown.
- <u>Distributed Computing</u> A calculator demonstrating remote service abstractions that let you 'export' a plain .NET object (PONO) via .NET Remoting, Web Services, or an EnterpriseService ServiceComponent. Corresponding client side proxies are also demonstrated.
- WCF Shows a WCF based calculator example that configures your WCF service via dependency injection and apply AOP advice.
- <u>Web Application SpringAir</u> -A ticket booking application that demonstrates the ASP.NET framework showing features such as DI for ASP.NET pages, data binding, validation, and localization.
- Web Development Introductory examples showing use of dependency injection and Spring's bi-directional data binding in ASP.NET.
- <u>Data Access</u> Demonstrates the ADO.NET framework showing how to simplify developing ADO.NET based data access layers.
- <u>Transaction Management</u>: Demonstrates the use of declarative transaction management for both local and distributed transaction in both .NET 1.1 and 2.0.

- AJAX : Demonstrates how to access a plain .NET object as a webservice in client side JavaScript
- NHibernate Northwind: Demonstrates use of Spring's NHibernate integration to simplify the use of NHibernate. Web tier is also included showing how to use the Open-Session In View approach to session management in the web tier.
- Quartz Quickstart Application that shows the use of Quartz.NET integration for scheduling.
- NMS Applicatoin demonstrating NMS helper classes.

2.6. License Information

Spring.NET is licensed according to the terms of the Apache License, Version 2.0. The full text of this license are available online at http://www.apache.org/licenses/LICENSE-2.0. You can also view the full text of the license in the license.txt file located in the root installation directory.

2.7. Support

Training and support are available through <u>SpringSource</u> in addition to the mailing lists and forums you can find on the main <u>Spring.NET</u> website.

Chapter 3. Background information

3.1. Inversion of Control

In early 2004, Martin Fowler asked the readers of his site: when talking about Inversion of Control: "the question, is what aspect of control are they inverting?". After talking about the term Inversion of Control Martin suggests renaming the pattern, or at least giving it a more self-explanatory name, and starts to use the term *Dependency Injection*. His article continues to explain some of the ideas behind this important software engineering principle.

Other references you may find useful are

- Wikipedia Article Dependency Injection
- CodeProject article Dependency Injection for Loose Coupling

Chapter 4. Migrating from 1.1 M2

4.1. Introduction

Several API changes were made after 1.1 M2 (before 1.1 RC1)due primarily by the need to refactor the code base to remove circular dependency cycles, which are now all removed. Class and schema name changes were also made to provide a more consistent naming convention across the codebase. As a result of these changes, you can not simply drop in the new .dlls as you may have done in previous release. This document serves as a high level guide to the most likely areas where you will need to make changes to either your configuration or your code.

The file, BreakingChanges-1.1.txt, in the root directory of the distribution contains the full listing of breaking changes made for RC1 and higher

4.2. Important Changes

This section covers the common areas were you will need to make changes in code/configuration when migration from M2 to RC1or higher.

4.2.1. Namespaces

Note: If you previously installed Spring .xsd files to your VS.NET installation directory, remove them manually, and copy over the new ones, which have the -1.1.xsd suffix.

The names of the section handlers to register custom schemas has changed, from ConfigParsersSectionHandler to

NamespaceParsersSectionHandler.

The target namespaces have changed, the 'directory' named /schema/ has been removed. For example, the target schema changed from http://www.springframework.net/schema/tx to

http://www.springframework.net/tx.

A typical declaration to use custom schemas within your configuration file looks like this

```
<objects xmlns='http://www.springframework.net'
    xmlns:db="http://www.springframework.net/database"
    xmlns:tx="http://www.springframework.net/tx"
    xmlns:aop="http://www.springframework.net/aop">
```

The class XmlParserRegistry was renamed to NamespaceParserRegistry.

Renamed Spring.Validation.ValidationConfigParser to

Spring. Validation. Config. Validation Namespace Par

Renamed from DatabaseConfigParser to DatabaseNamespaceParser

Renamed/Moved Remoting.RemotingConfigParser to Remoting.Config.RemotingNamespaceParser

A typical registration of custom parsers within your configuration file looks like

A manual registration would look like this

```
NamespaceParserRegistry.RegisterParser(typeof(AopNamespaceParNamespaceParserRegistry.RegisterParser(typeof(DatabaseNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParser(typeof(TxNamespaceParserRegistry.RegisterParserRegistry.RegisterParserRegistry.RegisterParserRegistry.RegisterParserRegistry.RegisterParserRegistry.RegisterParserRegistry.RegisterParserRegistry.RegisterParserRegistry.RegisterParserRegistry.RegisterParserRegistry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Registry.Reg
```

4.2.2. Core

Moved Spring.Util.DynamicReflection to Spring.Reflection.Dynamic

Moved TypeRegistry and related classes from Spring.Context.Support to Spring.Core.TypeResolution

Moved Spring.Objects.TypeConverters to Spring.Core.TypeConvesion

4.2.3. Web

Moved Spring. Web. Validation to Spring. Web. UI. Validation

4.2.4. Data

Changed schema to use 'provider' instead of 'dbProvider' element, usage is now <db:provider .../> and not <db:dbProvider .../>

Moved TransactionTemplate, TransactionDelegate and ITransactionCallback

from Spring.Data to Spring.Data.Support

Moved AdoTemplate, AdoAccessor, AdoDaoSupport,

RowMapperResultSetExtractor from Spring.Data to Spring.Data.Core

Moved AdoPlatformTransactionManager,

 $Service Domain Platform Transaction Manager, \ and \ Tx Scope Transaction Manager$

from Spring.Data to Spring.Data.Core

Spring Framework



Part I. Core Technologies

This initial part of the reference documentation covers all of those technologies that are absolutely integral to the Spring Framework.

Foremost amongst these is the Spring Framework's Inversion of Control (IoC) container. A thorough treatment of the Spring Framework's IoC container is closely followed by comprehensive coverage of Spring's Aspect-Oriented Programming (AOP) technologies. The Spring Framework has its own AOP framework, which is conceptually easy to understand, and which successfully addresses the 80% sweet spot of AOP requirements in enterprise programming.

The core functionality also includes an expression language for lightweight scripting and a ui-agnostic validation framework.

Finally, the adoption of the test-driven-development (TDD) approach to software development is certainly advocated by the Spring team, and so coverage of Spring's support for integration testing is covered (alongside best practices for unit testing). The Spring team have found that the correct use of IoC certainly does make both unit and integration testing easier (in that the presence of properties and appropriate constructors on classes makes them easier to wire together on a test without having to set up service locator registries and suchlike)... the chapter dedicated solely to testing will hopefully convince you of this as well.

- Chapter 5, The IoC container
- Chapter 6, The IObjectWrapper and Type conversion
- Chapter 7, Resources
- Chapter 8, Threading and Concurrency Support
- Chapter 9, Object Pooling
- Chapter 11, Expression Evaluation
- Chapter 10, Spring.NET miscellanea
- Chapter 12, Validation Framework
- Chapter 13, Aspect Oriented Programming with Spring.NET
- Chapter 14, Aspect Library
- Chapter 15, Common Logging

• Chapter 16, Testing

Chapter 5. The IoC container

5.1. Introduction

This chapter covers the Spring Framework's implementation of the Inversion of Control (IoC) ^[1] principle

The Spring.Core assembly provides the basis for the Spring.NET Inversion of Control container. The <code>IObjectFactory</code> interface provides an advanced configuration mechanism capable of managing objects of any nature. The <code>IApplicationContext</code> interface builds on top of the <code>IObjectFactory</code> (it is a sub-interface) and adds other functionality such as easier integration with Spring.NET's Aspect Oriented Programming (AOP) features, message resource handling (for use in internationalization), event propagation and application layer-specific context such as <code>WebApplicationContext</code> for use in web applications.

In short, the IObjectFactory provides the configuration framework and basic functionality, while the IApplicationContext adds more enterprise-centric functionality to it. The IApplicationContext is a complete superset of the IObjectFactory, and any description of IObjectFactory capabilities and behavior should be considered to apply to IApplicationContexts as well.

This chapter is divided into two parts, with the <u>first part</u> covering the basic principles that apply to both the <code>IObjectFactory</code> and <code>IApplicationContext</code>, with the <u>second part</u> covering those features that apply only to the <code>IApplicationContext</code> interface.

If you are new to Spring.NET or IoC containers in general, you may want to consider starting with <u>Chapter 33</u>, <u>IoC Quickstarts</u>, which contains a number of introductory level examples that actually demonstrate a lot of what is described in detail below. Don't worry if you don't absorb everything at once... those examples serve only to paint a picture of how Spring.NET hangs together in really broad brushstrokes. Once you have finished with those examples, you can come back to this section which will fill in all the fine detail.

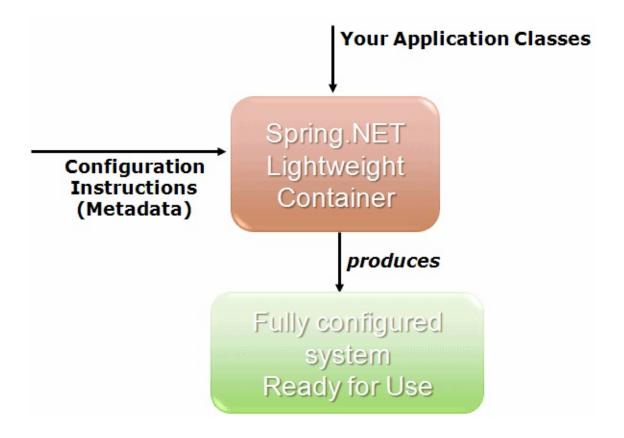
5.2. Basics - containers and objects

5.2.1. The container

The **IObjectFactory** is the actual representation of the Spring IoC container that is responsible for instantiating, configuring, and managing a number of objects.

The **IObjectFactory** interface is the central IoC container interface in Spring. Its responsibilities include instantiating or sourcing application objects, configuring such objects, and assembling the dependencies between these objects.

There are a number of implementations of the <code>IObjectFactory</code> interface that come supplied straight out-of-the-box with Spring. The most commonly used <code>IObjectFactory</code> implementation is the <code>XmlobjectFactory</code> class. This implementation allows you to express the objects that compose your application, and the doubtless rich interdependencies between such objects, in terms of <code>XML</code>. The <code>XmlobjectFactory</code> takes this <code>XML</code> configuration metadata and uses it to create a fully configured system or application. Interaction with the <code>IObjectFactory</code> interface is discussed in <code>Section 5.2.6</code>. "Using the container". Additional features offered by another implementation of <code>IObjectFactory</code>, the <code>IApplicationContext</code>, are discussed in <code>Section 5.10</code>. "The <code>IApplicationContext</code>".



5.2.1.1. Configuration metadata

As can be seen in the above image, the Spring IoC container consumes some form of configuration metadata; this configuration metadata is nothing more than how you (as an application developer) inform the Spring container as to how to "instantiate, configure, and assemble [the objects in your application]". This configuration metadata is typically supplied in a simple and intuitive XML format. When using XML-based configuration metadata, you write object definitions for those object that you want the Spring IoC container to manage, and then let the container do it's stuff.



Note

XML-based metadata is by far the most commonly used form of configuration metadata. It is not however the only form of configuration metadata that is allowed. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. Attribute based metadata will be part of an upcoming release and it is already part of the Spring Java

framework.

Spring configuration consists of at least one object definition that the container must manage, but typically there will be more than one object definition. When using XML-based configuration metadata, these object are configured as <object/> elements inside a top-level <objects/> element.

These object definitions correspond to the actual objects that make up your application. Typically you will have object definitions for your service layer objects, your data access objects (DAOs), presentation objects such as ASP.NET page instances, infrastructure objects such as NHibernate SessionFactories, and so forth. Typically one does not configure fine-grained domain objects in the container, because it is usually the responsibility of DAOs and business logic to create/load domain objects.

Find below an example of the basic structure of XML-based configuration metadata.

5.2.2. Instantiating a container

Instantiating a Spring IoC container is straightforward.

```
IObjectFactory factory = context;
```

You can also create an container by using a custom configuration section in the standard .NET application (or web) configuration file. Once the container has been created you may never need to explicitly interact with it again in your code, for example when configuring ASP.NET pages.

You may be wondering what the assembly URL is all about. The above example uses Spring.NET's <u>IResource</u> abstraction. The **IResource** interface provides a simple and uniform interface to a wide array of IO resources that can represent themselves as System. IO. Stream. An example for a file based resource, not using the URL syntax but an implementation of the IResource interface for file is shown below.

```
[C#]
IResource input = new FileSystemResource ("objects.xml");
IObjectFactory factory = new XmlObjectFactory(input);
```

These resources are most frequently files or URLs but can also be resources that have been embedded inside a .NET assembly. A simple URI syntax is used to describe the location of the resource, which follows the standard conventions for files, i.e. file://object.xml and other well known protocols such as http.

The following snippet shows the use of the URI syntax for referring to a that has been embedded inside a .NET assembly. resource assembly://<AssemblyName>/<NameSpace>/<Resource The **IResource** abstraction is explained further in <u>Section 7.1</u>, "Introduction".



To create an embedded resource using Visual Studio you must set the Build Action of the .xml configuration file to Embedded Resource in the file property editor. Also, you will need to explicitly rebuild the project containing the configuration file if it is the only change you make between successive builds. If using NAnt to build, add a <re>ources> section to the csc task. For example usage, look at the Spring.Core.Tests.build file included the distribution.

IApplicationContext The preferred create an

IObjectFactory is to use a custom configuration section in the standard .NET application configuration file (one of App.config or Web.config). A custom configuration section that creates the same IApplicationContext as the previous example is

```
<spring>
  <context type="Spring.Context.Support.XmlApplicationContex
     <resource uri="file://services.xml"/>
        <resource uri="assembly://MyAssembly/MyDataAccess/data-a
        </context>
</spring>
```

The context type (specified as the value of the type attribute of the context element) is wholly optional, and defaults to the Spring.Context.Support.XmlApplicationContext class, so the following XML snippet is functionally equivalent to the first.

```
<spring>
  <context>
    <resource uri="file://services.xml"/>
     <resource uri="assembly://MyAssembly/MyDataAccess/data-a
  </context>
</spring>
```

To acquire a reference to an **IApplicationContext** using a custom configuration section, one simply uses the following code;

```
IApplicationContext ctx = ContextRegistry.GetContext();
```

The **ContextRegistry** is used to both instantiate the application context and to perform service locator style access to other objects. (See Section 5.15, "Service Locator access" for more information). The glue that makes this possible is implementation of the Base Class Library (BCL) IConfigurationSectionHandler interface, namely the Spring.Context.Support.ContextHandler The class. handler class needs to be registered in the configSections section of the .NET configuration file as shown below.

```
<configSections>
  <sectionGroup name="spring">
        <section name="context" type="Spring.Context.Support.Cor
        </sectionGroup>
  </configSections>
```

This declaration now enables the use of a custom context section starting at the **spring** root element.

In some usage scenarios, user code will not have to explicitly instantiate an appropriate implementation of the <code>IObjectFactory</code> interface, since Spring.NET code will do it. For example, the ASP.NET web layer provides support code to load a Spring.NET <code>IApplicationContext</code> automatically as part of the normal startup process of an ASP.NET web application. Similar support for WinForms applications is being investigated.

While programmatic manipulation of IObjectFactory instances will be described later, the following sections will concentrate on describing the configuration of objects managed by IObjectFactory instances.

Spring.NET comes with an XSD schema to make the validation of the XML object definitions a whole lot easier. The XSD document is thoroughly documented so feel free to take a peek inside (see Appendix C. Spring.NET's spring-objects.xsd). The XSD is currently used in the implementation code to validate the XML document. The XSD schema serves a dual purpose in that it also facilitates the editing of XML object definitions inside an XSD aware editor (typically Visual Studio) by providing validation (and Intellisense support in the case of Visual Studio). You may wish to refer to Chapter 32, Visual Studio.NET Integration for more information regarding such integration.

Your XML object definitions can also be defined within the standard .NET configuration file registering application by the Spring.Context.Support.DefaultSectionHandler class as the configuration section handler for inline object definitions. This allows configure to completely one more IApplicationContext instances within a single standard .NET application configuration file as shown in the following example.

```
<configuration>
```

Other options available to structure the configuration files are described in Section 5.12.1, "Context Hierarchies" and Section 5.2.2.1, "Composing XML-based configuration metadata".

The **IApplicationContext** can be configured to register other resource handlers, custom parsers to integrate user-contributed XML schema into the object definitions section, type converters, and define type aliases. These features are discussed in section <u>Section 5.11</u>, "Configuration of IApplicationContext"

5.2.2.1. Composing XML-based configuration metadata

It is often useful to split up container definitions into multiple XML files. One way to then load an application context which is configured from all these XML fragments is to use the application context constructor which takes multiple resource locations. With an object factory, an object definition reader can be used multiple times to read definitions from each file in turn.

Generally, the Spring.NET team prefers the above approach, assembling individual files because it keeps container configuration files unaware of the fact that they are being combined with others. However, an alternate approach is to compose one XML object definition file using one or more occurrences of the

import element to load definitions from other files. Any import elements
must be placed before Object elements in the file doing the importing. Let's
look at a sample:

```
<objects xmlns="http://www.springframework.net">
    <import resource="services.xml"/>
        <import resource="resources/messageSource.xml"/>
        <import resource="/resources/themeSource.xml"/>
        <object id="object1" type="..."/>
        <object id="object2" type="..."/>
        </objects>
```

In this example, external object definitions are being loaded from 3 files, services.xml, messageSource.xml, and themeSource.xml. All location paths are considered relative to the definition file doing the importing, so Services.xml in this case must be the file same directory as doing the importing, messageSource.xml and themeSource.xml must be in a resources location below the location of the importing file. As you can see, a leading slash is actually ignored, but given that these are considered relative paths, it is probably better form not to use the slash at all. The contents of the files being imported must be fully valid XML object definition files according to the XSD, including the top level **Objects** element.

5.2.3. The Objects

A Spring IoC container manages one or more objects. these objects are created using the configuration metadata that has been supplied to the container (typically in the form of XML <object/> definitions).

Within the container itself, these object definitions are represented as IObjectDefinition objects, which contain (among other information) the following metadata:

• *A type name*: typically this is the actual implementation class of the object being defined..

- Object behavioral configuration elements, which state how the object should behave in the Spring.NET IoC container (i.e. prototype or singleton, lifecycle callbacks, and so forth)
- references to other objects which are needed for the object to do its work: these references are also called *collaborators* or *dependencies*.
- other configuration settings to set in the newly created object. An example would be the number of threads to use in an object that manages a worker thread pool, or the size limit of the pool.

The concepts listed above directly translate to a set of elements the object definition consists of. These elements are listed below, along with a link to further documentation about each of them.

Table 5.1. Object definition explanation

Feature	More info	
type	Section 5.2.4, "Object creation"	
id and name	Section 5.2.3.1, "Naming objects"	
singleton or prototype	Section 5.4, "Object Scopes"	
object properties	Section 5.3.1, "Injecting dependencies"	
constructor arguments	Section 5.3.1, "Injecting dependencies"	
autowiring mode	Section 5.3.6, "Autowiring collaborators"	
dependency checking mode	Section 5.3.7, "Checking for dependencies"	
initialization method	Section 5.6.1, "Lifecycle interfaces"	
destruction method	Section 5.6.1, "Lifecycle interfaces"	

Besides object definitions which contain information on how to create a specific object, certain <code>IObjectFactory</code> implementations also permit the registration of existing objects that have been created outside the factory (by user code). The <code>DefaultListableObjectFactory</code> class supports this through the <code>RegisterSingleton(..)</code> method. (Typical applications solely work with objects defined through metadata object definitions though.)

5.2.3.1. Naming objects

Every object has one or more ids (also called identifiers, or names; these terms refer to the same thing). These ids must be unique within the container the object is hosted in. An object will almost always have only one id, but if an object has more than one id, the extra ones can essentially be considered aliases.

When using XML-based configuration metadata, you use the 'id' or 'name' attributes to specify the object identifier(s). The 'id' attribute allows you to specify exactly one id, and as it is a real XML element ID attribute, the XML parser is able to do some extra validation when other elements reference the id; as such, it is the preferred way to specify an object id. However, the XML specification does limit the characters which are legal in XML IDs. This is usually not a constraint, but if you have a need to use one of these special XML characters, or want to introduce other aliases to the object, you may also or instead specify one or more object ids, separated by a comma (,), semicolon (;), or whitespace in the 'name' attribute.

Please note that you are not required to supply a name for a object. If no name is supplied explicitly, the container will generate a unique name for that object. The motivations for not supplying a name for a object will be discussed later (one use case is inner objects).

5.2.3.1.1. Aliasing objects

In an object definition itself, you may supply more than one name for the object, by using a combination of the id and name attributes as discussed in Section 5.2.3.1. "Naming objects". This approach to aliasing objects has some limitations when you would like to assemble the main application configuration file from multiple files. This usage pattern is common when each configuration file represents a logical layer or component within the application. In this case you may want to refer to a common object dependency using a name that is specific to each file. If the common object dependency is defined in the main application configuration file itself, then one can use the name element as an alias mechanism. However, if the main application configuration file should not be responsible for defining the common object dependency, since it logically 'belongs' to one of the other layers or components, you can not use the name

attribute to achieve this goal.

In this case, you can define an alias using an explicit alias element contained in the main application configuration file.

```
<alias name="fromName" alias="toName"/>
```

This allows an object named **fromName** to be referred to as **toName** across all application configuration files.

As a concrete example, consider the case where the configuration file 'a.xml' (representing component A) defines a connection object called componentA-connection. In another file, 'b.xml' (representing component B) would like to refer to the connection as componentB-connection. And the main application, MyApp, defines its own XML fragment to assembles the final application configuration from all three fragments and would like to refer to the connection as myApp-connection. This scenario can be easily handled by adding to the MyApp XML fragment the following standalone aliases:

```
<alias name="componentA-connection"
alias="componentB-connection"/>
<alias name="componentA-connection"
alias="myApp-connection"/>
```

Now each component and the main app can refer to the connection via a name that is unique and guaranteed not to clash with any other definition (effectively there is a namespace), yet they refer to the same object.

5.2.4. Object creation

An object definition essentially is a recipe for creating one or more objects. The container looks at the recipe for a named object when asked, and uses the configuration metadata encapsulated by that object definition to create (or acquire) an actual object.

If you are using XML-based configuration metadata, you can specify the type of object that is to be instantiated using the 'type' attribute of the <Object/> element. This 'type' attribute (which internally eventually boils down to being a Type property on a IObjectDefinition instance) is normally mandatory (see XXX "Instantiation using an instance factory method" and XXX "Object definition inheritance" for the two exceptions) and is used for one of two purposes. The type property specifies the

type of the object to be constructed in the common case where the container itself directly creates the object by calling its constructor reflectively (somewhat equivalent to C# code using the 'new' operator). In the less common case where the container invokes a Static, factory method on a class to create the object, the type property specifies the actual class containing the Static factory method that is to be invoked to create the object (the type of the object returned from the invocation of the Static factory method may be the same type or another type entirely, it doesn't matter).

5.2.4.1. Object creation via constructor invocation

When creating an object using the constructor approach, all normal classes are usable by and compatible with Spring. That is, the type being created does not need to implement any specific interfaces or be coded in a specific fashion. Just specifying the object type should be enough. However, depending on what type of IoC you are going to use for that specific object, you may need to create a default constructor (i.e. a constructor that has no parameters) in the source code definition of your class.

The XmlobjectFactory implementation of the IobjectFactory interface can consume object definitions that have been defined in XML, for example...

```
<object id="exampleObject" type="Examples.ExampleObject, ExampleObject, Exam
```

The mechanism for supplying arguments to the constructor (if required), or setting properties of the object instance after it has been constructed, is described shortly.

This XML fragment describes an object definition that will be identified by the exampleObject of which will be of name, instances the Examples. ExampleObject type that has been compiled into the ExamplesLibrary assembly. Take special note of the structure of the **type** attribute's value... the namespace-qualified name of the class is specified, followed by a comma, followed by (at a bare minimum) the name of the assembly that contains the class. preceding example, In the ExampleObject class is defined in the Examples namespace, and it has been compiled into the ExamplesLibrary assembly.

The name of the assembly that contains the type *must* be specified in the type attribute. Furthermore, it is recommended that you specify the fully qualified assembly name ^[2] in order to guarantee that the type that Spring.NET uses to instantiate your object (s) is indeed the one that you expect. Usually this is only an issue if you are using classes from (strongly named) assemblies that have been installed into the Global Assembly Cache (GAC).

If you have defined nested classes use the addition symbol, +, to reference the nested class. For example, if the class <code>Examples.ExampleObject</code> had a nested class <code>Person</code> the XML declaration would be

```
<object id="exampleObject" type="Examples.ExampleObject+Pers</pre>
```

If you are defining classes that have been compiled into assemblies that are available to your application (such as the bin directory in the case of ASP.NET applications) via the standard assembly probing mechanisms, then you can name of assembly specify simply the the ExamplesLibrary. Data)... this way, when (or if) the assemblies used by your application are updated, you won't have to change the value of every <object/> definition's type attribute to reflect the new version number (if the version number has changed)... Spring.NET will automatically locate and use the newer versions of your assemblies (and their attendant classes) from that point forward.

5.2.4.2. Object creation via a static factory method

When defining an object which is to be created using a static factory method, along with the type attribute which specifies the type containing the static factory method, another attribute named factory-method is needed to specify the name of the factory method itself. Spring.NET expects to be able to call this method (with an optional list of arguments as described later) and get back a live object, which from that point on is treated as if it had been created normally via a constructor. One use for such an object definition is to call static factories in legacy code.

Following is an example of an object definition which specifies that the object is to be created by calling a factory-method. Note that the definition does not specify the type (class) of the returned object, only the type containing the factory method. In this example, CreateInstance must be a static method.

```
<object id="exampleObject"
          type="Examples.ExampleObjectFactory, ExamplesLibrary"
          factory-method="CreateInstance"/>
```

The mechanism for supplying (optional) arguments to the factory method, or setting properties of the object instance after it has been returned from the factory, will be described shortly.

5.2.4.3. Object creation via an instance factory method

In a fashion similar to instantiation using a static factory method, instantiation using an instance factory method is where a non-static method of an existing object from the container is invoked to create the new object. To use this mechanism, the 'type' attribute must be left empty, and the 'factory-Object' attribute must specify the name of an object in the current (or parent/ancestor) container that contains the instance method that is to be invoked to create the object. The name of the factory method itself should still be set via the 'factory-method' attribute.

```
<!-- the factory object, which contains an instance method o
<object id="exampleFactory" type="...">
    <!-- inject any dependencies required by this object -->
</object>
<!-- the object that is to be created by the factory object
<object id="exampleObject"
    factory-method="CreateInstance"
    factory-object="exampleFactory"/>
```

Although the mechanisms for setting object properties are still to be discussed, one implication of this approach is that the factory object itself can be managed and configured via Dependency Injection, by the container.



When the Spring documentation makes mention of a 'factory object', this will be a reference to an object that is configured in the Spring

container that will create objects via an instance or static factory method. When the documentation mentions a <code>IFactoryObject</code> (notice the capitalization) this is a reference to a Spring-specific <code>IFactoryObject</code>.

5.2.5. Object creation of generic types

Generic types can also be created in much the same manner an non-generic types.

5.2.5.1. Object creation of generic types via constructor invocation

The following examples shows the definition of simple generic types and how they can be created in Spring's XML based configuration file.

```
namespace GenericsPlay
{
    public class FilterableList<T>
        private List<T> list;
        private String name;
        public List<T> Contents
            get { return list; }
            set { list = value; }
        }
        public String Name
            get { return name; }
            set { name = value; }
        }
        public List<T> ApplyFilter(string filterExpression)
        {
            /// should really apply filter to list ;)
            return new List<T>();
        }
```

```
}
```

The XML configuration to create and configure this object is shown below

There are a few items to note in terms how to specify a generic type. First, the left bracket that specifies the generic type, i.e. <, is replaced with the string < due to XML escape syntax for the less than symbol. Yes, we all realize this is less than ideal from the readability point of view. Second, the generic type arguments can not be fully assembly qualified as the comma is used to separate generic type arguments. Alternative characters used to overcome the two quirks can be implemented in the future but so far, all proposals don't seem to help clarify the text. The suggested solution to improve readability is to use type aliases as shown below

So that instead of something like this

It can be shortened to

or even shorter

Refer to <u>Section 5.11</u>, "Configuration of <u>IApplicationContext</u>" for additional information on using type aliases.

5.2.5.2. Object creation of generic types via static factory method

The following classes are used to demonstrate the ability to create instances of generic types that themselves are created via a static generic factory method.

```
public class TestGenericObject<T, U>
{
   public TestGenericObject()
   {
    }
   private IList<T> someGenericList = new List<T>();

   private IDictionary<string, U> someStringKeyedDictionary
        new Dictionary<string, U>();

   public IList<T> SomeGenericList
   {
      get { return someGenericList; }
      set { someGenericList = value; }
   }

   public IDictionary<string, U> SomeStringKeyedDictionary
   {
      get { return someStringKeyedDictionary; }
      set { someStringKeyedDictionary = value; }
   }
}
```

The accompanying factory class is

```
public class TestGenericObjectFactory
{
    public static TestGenericObject<V, W> StaticCreateInstar
    {
        return new TestGenericObject<V, W>();
```

```
public TestGenericObject<V, W> CreateInstance<V, W>()
{
    return new TestGenericObject<V, W>();
}
```

The XML snippet to create an instance of TestGenericObject where V is a List of integers and W is an integer is shown below

The StaticCreateInstance method is responsible for instantiating the object that will be associated with the id 'myTestGenericObject'.

5.2.5.3. Object creation of generic types via instance factory method

Using the class from the previous example the XML snippet to create an instance of a generic type via an instance factory method is shown below

This creates an instance of TestGenericObject<List<int>, int>

5.2.6. Using the container

An IApplicationContext is essentially nothing more than the interface for an advanced factory capable of maintaining a registry of different objects and their dependencies. The IApplicationContext enables you to read object definitions and access them. You create one and read in some

object definition in the XML format as follows:

IApplicationContext context = new XmlApplicationContext("fil

Basically that is all there is to it. Using GetObject(string) or the indexer [string], you can retrieve instances of your object; the client-side view of the IApplicationContext is simple. The IApplicationContext interface has just a few other methods related to finding objects in the contianer, but ideally your application code should never use them... indeed, your application code should have no calls to the GetObject(string) method at all, and thus no dependency on Spring APIs at all.

5.3. Dependencies

Your typical enterprise application is not made up of a single object. Even the simplest of applications will no doubt have at least a handful of objects that work together to present what the end-user sees as a coherent application. This next section explains how you go from defining a number of object definitions that stand-alone, each to themselves, to a fully realized application where objects work (or collaborate) together to achieve some goal (usually an application that does what the end-user wants).

5.3.1. Injecting dependencies

The basic principle behind Dependency Injection (DI) is that objects define their dependencies (that is to say the other objects they work with) only through constructor arguments, arguments to a factory method, or properties which are set on the object instance after it has been constructed or returned from a factory method. Then, it is the job of the container to actually inject those dependencies when it creates the object. This is fundamentally the inverse, hence the name Inversion of Control (IoC), of the object itself being in control of instantiating or locating its dependencies on its own using direct construction of classes, or something like the Service Locator pattern.

It becomes evident upon usage that code gets much cleaner when the DI principle is applied, and reaching a higher grade of decoupling is much easier when objects do not look up their dependencies, but are provided with them (and additionally do not even know where the dependencies are located and of what concrete class they are). DI exists in two major variants, namely Constructor Injection and Setter Injection.

5.3.1.1. Constructor Injection

Constructor-based DI is effected by invoking a constructor with a number of arguments, each representing a dependency. Additionally, calling a static factory method with specific arguments to construct the object, can be considered almost equivalent, and the rest of this text will consider arguments to a constructor and arguments to a static factory method similarly. Find below an example of a class that could only be dependency injected using constructor injection. Notice that there is nothing special about this class.

```
public class SimpleMovieLister
{
    // the SimpleMovieLister has a dependency on a MovieFinder
    private IMovieFinder movieFinder;

    // a constructor so that the Spring container can 'inject'
    public MovieLister(IMovieFinder movieFinder)
    {
        this.movieFinder = movieFinder;
    }

    // business logic that actually 'uses' the injected IMovie
}
```

5.3.1.1.1. Constructor Argument Resolution

Constructor argument resolution matching occurs using the argument's type. If there is no potential for ambiguity in the constructor arguments of a object definition, then the order in which the constructor arguments are defined in a object definition is the order in which those arguments will be supplied to the appropriate constructor when it is being instantiated. Consider the following class:

```
namespace X.Y
{
    public class Foo
    {
        public Foo(Bar bar, Baz baz)
        {
            // ...
        }
    }
}
```

There is no potential for ambiguity here (assuming of course that Bar and Baz classes are not related in an inheritance hierarchy). Thus the following configuration will work just fine, and you do not need to specify the constructor

argument indexes and / or types explicitly.

When another object is referenced, the type is known, and matching can occur (as was the case with the preceding example). When a simple type is used, such as <value>true<value>, Spring cannot determine the type of the value, and so cannot match by type without help. Consider the following class:

5.3.1.1.1. Constructor Argument Type Matching

The above scenario *can* use type matching with simple types by explicitly specifying the type of the constructor argument using the **type** attribute. For example:

The type attribute specifies the System. Type of the constructor argument, such as System.Int32. Alias' are available to for common simple types (and their array equivalents). These alias' are...

Table 5.2.	Type	aliases
-------------------	------	---------

Type	Alias'	Array Alias'
System.Char	char, Char	char[], Char()
System.Int16	short, Short	short[], Short()
System.Int32	int, Integer	int[], Integer()
System.Int64	long, Long	long[], Long()
System.UInt16	ushort	ushort[]
System.UInt32	uint	uint[]
System.UInt64	ulong	ulong[]
System.Float	float, Single	float[], Single()
System.Double	double, Double	double[], Double()
System.Date	date, Date	date[], Date()
System.Decimal	decimal, Decimal	decimal[], Decimal()
System.Bool	bool, Boolean	bool[], Boolean()
System.String	string, String	string[], String()

5.3.1.1.1.2. Constructor Argument Index

Constructor arguments can have their index specified explicitly by use of the index attribute. For example:

```
<constructor-arg index="1" value="42"/>
</object>
```

As well as solving the ambiguity problem of multiple simple values, specifying an index also solves the problem of ambiguity where a constructor may have two arguments of the same type. Note that the *index* is 0 based.

5.3.1.1.3. Constructor Arguments by Name

Constructor arguments can also be specified by name by using the name attribute of the <constructor-arg> element.

5.3.1.2. Setter Injection

Setter-based DI is realized by calling setter methods on your objects after invoking a no-argument constructor or no-argument static factory method to instantiate your object.

Find below an example of a class that can only be dependency injected using pure setter injection.

```
public class MovieLister
{
   private IMovieFinder movieFinder;

   public IMovieFinder MovieFinder
   {
       set
       {
            movieFinder = value;
       }
   }

// business logic that actually 'uses' the injected IMovie
}
```

The IObjectFactory supports both of these variants for injecting dependencies into objects it manages. (It in fact also supports injecting setter-based dependencies after some dependencies have already been supplied via the constructor approach.) The configuration for the dependencies comes in the form of the IObjectDefinition which is used class, together TypeConverters to know how to convert properties from one format to another. However, most users of Spring.NET will not be dealing with these classes directly (that is programatically), but rather with an XML definition file which will be converted internally into instances of these classes, and used to load an entire Spring IoC container instance.

Object dependency resolution generally happens as follows:

- 1. The IObjectFactory is created and initialized with a configuration which describes all the objects. Most Spring.NET users use an IObjectFactory or IApplicationContext variant that supports XML format configuration files.
- 2. Each object has dependencies expressed in the form of properties, constructor arguments, or arguments to the static-factory method when that is used

instead of a normal constructor. These dependencies will be provided to the object, when the object is actually created.

3. Each property or constructor argument is either an actual definition of the

Constructor- or Setter-based DI?

Spring team generally advocates the usage of setter injection, since a large number of constructor arguments can get unwieldy, especially when some properties are optional. The presence of setter methods also makes objects of that amenable to being re-configured (or re-injected) at some later time (for management purposes).

Constructor-injection is favored by some purists though and with good reason. Supplying all of an object's dependencies means that that object is never returned to client (calling) code in a less than totally initialized state. The flip side is that the object becomes less amenable to re-configuration (or reinjection).

There is no hard and fast rule here. Use whatever type of DI makes the most sense for a particular class; sometimes, when dealing with third party classes to which you do not have the source, the choice will already have been made for you - a legacy class may not expose any setter methods, and so constructor injection will be the only type of DI available to you.

Since you can mix both, Constructor- and Setter-based DI, it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

- value to set, or a reference to another object in the container.
- 4. Each property or constructor argument which is a value must be able to be converted from whatever format it was specified in, to the actual <code>System.Type</code> of that property or constructor argument. By default Spring.NET can convert a value supplied in string format to all built-in types, such as <code>int</code>, <code>long</code>, <code>string</code>, <code>bool</code>, etc. Spring.NET uses <code>TypeConverter</code> definitions to be able to convert string values to other, arbitrary types. Refer to <code>Section 6.3</code>, "Type conversion" for more information regarding type conversion, and how you can design your classes to be convertible by <code>Spring.NET</code>.

The Spring container validates the configuration of each object as the container is created, including the validation that properties which are object references are actually referring to valid object. However, the object properties themselves are not set until the object is actually created. For those object that defined as singletons and set to be pre-instantiated (such as singleton object in an <code>IApplicationContext</code>), creation happens at the time that the container is created, but otherwise this is only when the object is requested. When an object actually has to be created, this will potentially cause a graph of other objects to be created, as its dependencies and its dependencies' dependencies (and so on) are created and assigned.

You can generally trust Spring.NET to do the right thing. It will detect configuration issues, such as references to non-existent object definitions and circular dependencies, at container load-time. It will actually set properties and resolve dependencies as late as possible, which is when the object is actually created. This means that a Spring container which has loaded correctly can later generate an exception when you request an object if there is a problem creating that object or one of its dependencies. This could happen if the object throws an exception as a result of a missing or invalid property, for example. This potentially delayed visibility of some configuration issues is why

Circular Dependencies

If you are using predominantly constructor injection it is possible to write and configure your classes and objects such that an unresolvable circular dependency scenario is created.

Consider the scenario where you have class A, which requires an instance of class B to be provided via constructor injection, and class B, which requires an instance of class A to be provided via constructor injection. If you configure object for classes A and B to be injected into each other, the Spring IoC container will detect this circular reference at runtime, and throw a

IApplicationContext by default pre-instantiates singleton objects. At the cost of some upfront time and memory to create these objects before they are actually needed, you find out about configuration issues when the IApplicationContext is created, not later. If you wish, you can still override this default behavior and set any of these singleton objects to lazy-load (not be preinstantiated)

If no circular dependencies are involved (see sidebar for а discussion of circular dependencies), when one or more collaborating objects are being injected into a dependent object, each collaborating object is totally configured prior to being passed (via one of the DI flavors) to the dependent

ObjectCurrentlyInCre

One possible solution to this issue is to edit the source code of some of your classes to be configured via setters instead of via constructors. Another solution is not to use constructor injection and stick to setter injection only. In other words, while it should generally be avoided in all but the rarest of circumstances, it is possible to configure circular dependencies with setter injection.

Unlike the typical case (with no circular dependencies), a circular dependency between object A and object B will force one of the objects to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).

object. This means that if object A has a dependency on object B, the Spring IoC container will *totally* configure object B prior to invoking the setter method on object A; you can read *'totally configure'* to mean that the object will be instantiated (if not a pre-instantiated singleton), all of its dependencies will be set, and the relevant lifecycle methods (such as a configured init method or the IIntializingObject callback method) will all be invoked.

5.3.1.3. Some examples

First, an example of using XML-based configuration metadata for setter-based DI. Find below a smallpart of a Spring XML configuration file specifying some object definitions.

```
[C#]
public class ExampleObject
{
   private AnotherObject objectOne;
   private YetAnotherObject objectTwo;
   private int i;

public AnotherObject ObjectOne
   {
     set { this.objectOne = value; }
}

public YetAnotherObject ObjectTwo
   {
     set { this.objectTwo = value; }
}

public int IntegerProperty
   {
     set { this.i = value; }
}
```

As you can see, setters have been declared to match against the properties specified in the XML file. Find below an example of using constructor-based DI.

```
[Visual Basic.NET]
Public Class ExampleObject
```

```
Private myObjectOne As AnotherObject
Private myObjectTwo As YetAnotherObject
Private i As Integer

Public Sub New (
    anotherObject as AnotherObject,
    yetAnotherObject as YetAnotherObject,
    i as Integer)

myObjectOne = anotherObject
myObjectTwo = yetAnotherObject
Me.i = i
End Sub
End Class
```

As you can see, the constructor arguments specified in the object definition will be used to pass in as arguments to the constructor of the ExampleObject.

Now consider a variant of this where instead of using a constructor, Spring is told to call a static factory method to return an instance of the object

```
[C#]
public class ExampleFactoryMethodObject
{
   private AnotherObject objectOne;
   private YetAnotherObject objectTwo;
   private int i;

// a private constructor
   private ExampleFactoryMethodObject()
   {
   }
}
```

Note that arguments to the static factory method are supplied via constructor-arg elements, exactly the same as if a constructor had actually been used. These arguments are optional. Also, it is important to realize that the type of the class being returned by the factory method does not have to be of the same type as the class which contains the static factory method, although in this example it is. An instance (non-static) factory method, mentioned previously, would be used in an essentially identical fashion (aside from the use of the factory-object attribute instead of the type attribute), so will not be detailed here.

Note that Setter Injection and Constructor Injectionare not mutually exclusive. It is perfectly reasonable to use both for a single object definition, as can be seen in the following example:

```
[C#]
public class MixedIocObject
{
```

```
private AnotherObject objectOne;
private YetAnotherObject objectTwo;
private int i;

public MixedIocObject (AnotherObject obj)
{
   this.objectOne = obj;
}

public YetAnotherObject ObjectTwo
{
   set { this.objectTwo = value; }
}

public int IntegerProperty
{
   set { this.i = value; }
}
```

5.3.2. Dependencies and configuration in detail

As mentioned in the previous section, object properties and constructor arguments can be defined as either references to other managed objects (collaborators), or values defined inline. Spring's XML-based configuration metadata supports a number of sub-element types within its property/> and constructor-arg/> elements for this purpose.a

5.3.2.1. Straight values (primitives, Strings, etc.)

The <value/> element specifies a property or constructor argument as a representation. human-readable string As mentioned previously, TypeConverter instances are used to convert these string values from a System.String to the actual property or argument type. Custom TypeConverter implementations in the Spring.Objects.TypeConverters namespace are used to augment the functionality offered by the .NET BCL's default TypeConverter implementations.

In the following example, we use a SqlConnection from the

System.Data.SqlClient namespace of the BCL. This class (like many other existing classes) can easily be used in a Spring.NET object factory, as it offers a convenient public property for configuration of its ConnectionString property.

5.3.2.1.1. The idref element

An idref element is simply a shorthand and error-proof way to set a property to the String *id* or *name* of another object in the container.

This is exactly equivalent at runtime to the following fragment:

The main reason the first form is preferable to the second is that using the <code>idref</code> tag will allow Spring.NET to validate at deployment time that the other object actually exists. In the second variation, the class that is having its <code>targetName</code> property injected is forced to do its own validation, and that will only happen when that class is actually instantiated by the container, possibly long after the container is actually up and running.

Additionally, if the object being referred to is in the same actual XML file, and the object name is the object id, the local attribute may be used, which will allow the XML parser itself to validate the object name even earlier, at parse time.

```
< idref local="theTargetObject"/>
```

5.3.2.1.2. Whitespace Handling

Usually all leading and trailing whitespaces are trimmed from a <value /> element's text. In some cases it is necessary to maintain whitespaces exactly as they are written into the xml element. The parser does understand the xml:space attribute in this case:

```
< value xml:space="preserve"> &#x000a;&#x000d;&#x0009;</val
</pre>
```

The above configuration will result in the string " \n\r\t". Note, that you don't have to explicitely specify the 'xml' namespace on top of your configuration.

5.3.2.2. Referring to collaborating objects

The ref element is the final element allowed inside a property definition element. It is used to set the value of the specified property to be a reference to another object managed by the container, a collaborator, so to speak. As you saw in the previous example to set collection properties, we used the SqlConnection instance from the initial example as a collaborator and

specified it using a <ref object/> element. As mentioned in a previous section, the referred-to object is considered to be a dependency of the object who's property is being set, and will be initialized on demand as needed (if it is a singleton object it may have already been initialized by the container) before the property is set. All references are ultimately just a reference to another object, but there are 3 variations on how the id/name of the other object may be specified, which determines how scoping and validation is handled.

Specifying the target object by using the <code>Object</code> attribute of the <code>ref</code> tag is the most general form, and will allow creating a reference to any object in the same <code>IObjectFactory/IApplicationContext</code> (whether or not in the same <code>XML</code> file), or parent <code>IObjectFactory/IApplicationContext</code>. The value of the <code>Object</code> attribute may be the same as either the <code>id</code> attribute of the target object, or one of the values in the <code>name</code> attribute of the target object.

```
<ref object="someObject"/>
```

Specifying the target object by using the <code>local</code> attribute leverages the ability of the XML parser to validate XML id references within the same file. The value of the <code>local</code> attribute must be the same as the <code>id</code> attribute of the target object. The XML parser will issue an error if no matching element is found in the same file. As such, using the local variant is the best choice (in order to know about errors are early as possible) if the target object is in the same XML file.

```
<ref local="someObject"/>
```

Specifying the target object by using the parent attribute allows a reference to be created to an object that is in a parent IObjectFactory (orIApplicationContext) of the current IObjectFactory (or IApplicationContext). The value of the parent attribute may be the same as either the id attribute of the target object, or one of the values in the name attribute of the target object, and the target object must be in a parent IObjectFactory or IApplicationContext of the current one. The main use of this object reference variant is when there is a need to wrap an existing object in a parent context with some sort of proxy (which may have the

same name as the parent), and needs the original object so it may wrap it.

```
<ref parent="someObject"/>
```

5.3.2.3. Inline objects

An **object** element inside the **property** element is used to define an object value inline, instead of referring to an object defined elsewhere in the container. The inline object definition does not need to have any id or name defined (indeed, if any are defined, they will be ignored).

5.3.2.4. Setting collection values

The list, set, name-values and dictionary elements allow properties and arguments of the type IList, ISet, NameValueCollection and IDictionary, respectively, to be defined and set.

```
</list>
      </property>
      <!--
      results in a call to the setter of the SomeDictionary
      cproperty name="SomeDictionary">
          <dictionary>
              <entry key="a string => string entry" value="j
              <entry key-ref="myKeyObject" value-ref="myConr</pre>
          </dictionary>
      </property>
      <!--
      results in a call to the setter of the SomeNameValue
      cproperty name="SomeNameValue">
          <name-values>
              <add key="HarryPotter" value="The magic proper
              <add key="JerrySeinfeld" value="The funny (to
          </name-values>
      </property>
      <!--
      results in a call to the setter of the SomeSet (Spring
      property name="someSet">
          <set>
              <value>just some string</value>
              <ref object="myConnection"/>
          </set>
      </property>
  </object>
</objects>
```

Many classes in the BCL expose only read-only properties for collection classes. When Spring.NET encounters a read-only collection, it will configure the collection by using the getter property to obtain a reference to the collection class and then proceed to add the additional elements to the existing collection. This results in an additive behavior for collection properties that are exposed in this manner.

Note that the value of a Dictionary entry, or a set value, can also again be any of the elements:

The shortcut forms for value and references are useful to reduce XML verbosity when setting collection properties. See <u>Section 5.3.2.8</u>, "Value and ref shortcut forms" for more information.

5.3.2.5. Setting generic collection values

Spring supports setting values for classes that expose properties based on the generic collection interfaces IList<T> and IDictionary<TKey, TValue>. The type parameter for these collections is specified by using the XML attribute element-type for IList<T> and the XML attributes value-type IDictionary<TKey, key-type and for TValue>. The values of the collection are automatically converted from a string to the appropriate type. If you are using your own user-defined type as a generic type parameter you will likely need to register a custom type converter. Refer to Section 5.5, "Type conversion" for more information. The implementations of IList<T> and IDictionary<TKey, TValue> that is created are System.Collections.Generic.List and System.Collections.Generic.Dictionary.

The following class represents a lottery ticket and demonstrates how to set the values of a generic IList.

```
public class LotteryTicket {
  List<int> list;

  DateTime date;

public List<int> Numbers {
    set { list = value; }
    get { return list; }
}

public DateTime Date {
    get { return date; }
    set { date = value; }
}
```

```
}
}
```

The XML fragment that can be used to configure this class is shown below

The following shows the definition of a more complex class that demonstrates the use of generics using the Spring.Expressions.IExpression interface as the generic type parameter for the IList element-type and the value-type for IDictionary. Spring.Expressions.IExpression has an associated type converter, Spring.Objects.TypeConverters.ExpressionConvert that is already pre-registered with Spring.

```
public class GenericExpressionHolder
{
    private System.Collections.Generic.IList<IExpression
    private System.Collections.Generic.IDictionary<strir
    public System.Collections.Generic.IList<IExpression>
    {
        set { this.expressionsList = value; }
    }
    public System.Collections.Generic.IDictionary<string
    {
        set { System.Collections.Generic.IDictionary<string }
}</pre>
```

```
set { this.expressionsDictionary = value; }
}

public IExpression this[int index]
{
    get
    {
        return this.expressionsList[index];
    }
}

public IExpression this[string key]
{
    get { return this.expressionsDictionary[key]; }
}
```

An example XML configuration of this class is shown below

```
<object id="genericExpressionHolder"</pre>
    type="Spring.Objects.Factory.Xml.GenericExpressionHolder
    Spring.Core.Tests">
    property name="ExpressionsList">
        <list element-type="Spring.Expressions.IExpression,</pre>
            <value>1 + 1</value>
            <value>date('1856-7-9').Month</value>
            <value>'Nikola Tesla'.ToUpper()</value>
            <value>DateTime.Today > date('1856-7-9')</value>
        </list>
    </property>
    property name="ExpressionsDictionary">
        <dictionary key-type="string" value-type="Spring.Exp</pre>
            <entry key="zero">
                <value>1 + 1</value>
            </entry>
            <entry key="one">
                <value>date('1856-7-9').Month
            </entry>
            <entry key="two">
                <value>'Nikola Tesla'.ToUpper()</value>
            </entry>
            <entry key="three">
```

5.3.2.6. Setting null values

The <null> element is used to handle null values. Spring.NET treats empty arguments for properties and constructor arguments as empty String instances. The following configuration demonstrates this behaviour...

This results in the email property being set to the empty string value (""), in much the same way as can be seen in the following snippet of C# code...

```
exampleObject.Email = "";
```

The special <null/> element may be used to indicate a null value; to wit...

```
<object type="Examples.ExampleObject, ExamplesLibrary">
```

This results in the email property being set to **null**, again in much the same way as can be seen in the following snippet of C# code...

```
exampleObject.Email = null;
```

5.3.2.7. Setting indexer properties

An indexer lets you set and get values from a collection using a familiar bracket

[] notation. Spring's XML configuration supports the setting of indexer properties. Overloaded indexers as well as multiparameter indexers are also supported. The property expression parser described in Chapter 11, Expression
Evaluation is used to perform the type conversion of the indexer name argument from a string in the XML file to a matching target type. As an example consider the following class

```
public class Person
{
    private IList favoriteNames = new ArrayList();
    private IDictionary properties = new Hashtable();
    public Person()
        favoriteNames.Add("p1");
        favoriteNames.Add("p2");
    }
    public string this[int index]
        get { return (string) favoriteNames[index]; }
        set { favoriteNames[index] = value; }
    }
    public string this[string keyName]
        get { return (string) properties[keyName]; }
        set { properties.Add(keyName, value); }
    }
}
```

The XML configuration snippet to populate this object with data is shown below



The use of the property expression parser in Release 1.0.2 changed how you cor following section describes this usage.

The older style configuration uses the following syntax

You can also change the name used to identify the indexer by adorning your ind the attribute <code>[IndexerName("MyItemName")]</code>. You would then use t to configure the first element of that indexer.

There are some limitations to be aware in the older indexer configuration. The inparameter that is convertible from a string to the indexer parameter type. Also, r supported. You can get around that last limitation currently if you use the Indexe

5.3.2.8. Value and ref shortcut forms

There are also some shortcut forms that are less verbose than using the full value and ref elements. The property, constructor-arg, and entry elements all support a value attribute which may be used instead of embedding a full value element. Therefore, the following:

are equivalent to:

```
<entry key="myKey" value="hello"/>
```

In general, when typing definitions by hand, you will probably prefer to use the less verbose shortcut form.

The property and constructor-arg elements support a similar shortcut ref attribute which may be used instead of a full nested ref element. Therefore, the following...

```
<property name="myProperty">
    <ref object="anotherObject"/>
    </property>

<constructor-arg index="0">
        <ref object="anotherObject"/>
        </constructor-arg>
```

is equivalent to...



Note

The shortcut form is equivalent to a <ref object="xxx"> element; there is no shortcut for either the <ref local="xxx"> or <ref parent="xxx"> elements. For a local or parent ref, you must still use the long form.

Finally, the entry element allows a shortcut form the specify the key and/or value of a dictionary, in the form of key/key-ref and value/value-ref attributes. Therefore, the following

```
<entry>
     <key>
          <ref object="MyKeyObject"/>
          </key>
          <ref object="MyValueObject"/>
          </entry>
```

Is equivalent to:

```
<entry key-ref="MyKeyObject" value-ref="MyValueObject"/>
```

As mentioned previously, the equivalence is to <ref object="xxx"> and not the local or parent forms of object references.

5.3.2.9. Compound property names and Spring expression references

Note that compound or nested property names are perfectly legal when setting object properties. Property names are interpreted using the <u>Spring Expression Language</u> (SpEL) and therefore can leverage its many features to set property names. For example, in this object definition a simple nested property name is configured

As an example of some alternative ways to declare the property name, you can use SpEL's support for indexers to configure a Dictionary key value pair as an alternative to the nested **<dictionary>** element. More importantly, you can use the 'expression' element to refer to a Spring expression as the value of the property. Simple examples of this are shown below

Using SpEL's support for method evaluation, you can easily call static method on various helper classes in your XML configuration.

5.3.3. Declarative Event Listener Registration

In C# events are built right into the language thanks to the event keyword. Under the scenes, events are essentially a shorthand notation for delegates with some additional guidelines as to what the parameters to an event handler method should be (i.e. a sender System.Object and an

System. EventArgs object).

```
public class EventSource
public event EventHandler Click;
```

In use, .NET events are combined with one or more event handler methods. Each handler method is programmatically added, or removed, from the event and corresponds to an object's method that should be invoked when a particular event occurs. When more than one handler method is added to an event, then each of the registered methods will be invoked in turn when an event occurs.

```
TestObject source = new TestObject();
TestEventHandler eventListener1 = new TestEventHandler();
TestEventHandler eventListener2 = new TestEventHandler();
source.Click += eventListener1.HandleEvent; // Adding the fisource.Click += eventListener2.HandleEvent; // Adding a seconsource.OnClick(); // First eventListener1.HandleEvent is invented.
```

When OnClick() is invoked, the event is fired.

```
public void OnClick()
{
   if (Click != null)
    {
     Click(this, EventArgs.Empty); // Fire the event off to }
}
```

One of the not so nice things about using events is that, without employing late binding, you declare the objects that are registered with a particular event programmatically. Spring .NET offers a way to declaratively register your handler methods with particular events using the listener element inside your <object> elements.

5.3.3.1. Declarative event handlers

Rather than having to specifically declare in your code that you are adding a

method to be invoked on an event, using the tener> element you can register a plain object's methods with the corresponding event declaratively in your application configuration.

Using the listener element you can:

- Configure a method to be invoked when an event is fired.
- Register a collection of handler methods based on a regular expression.
- Register a handler method against an event name that contains a regular expression.

5.3.3.2. Configuring a method to be invoked when an event is fired

The same event registration in the example above can be achieved using configuration using the listener> element.

In this case the two different objects will have their <code>HandleEvent</code> method invoked, as indicated explicitly using the <code>method</code> attribute, when a <code>Click</code> event, as specified by the <code>event</code> attribute, is triggered on the object referred to by the <code>ref</code> element.

5.3.3.3. Registering a collection of handler methods based on a regular expression

Regular expressions can be employed to wire up more than one handler method to an object that contains one or more events.

Here all the eventListener's handler methods that begin with 'Handle', and that have the corresponding two parameters of a System.Object and a System.EventArgs, will be registered against all events exposed by the Source object.

You can also use the name of the event in regular expression to filter your handler methods based on the type of event triggered.

5.3.3.4. Registering a handler method against an event name that contains a regular expression

Finally, you can register an object's handler methods against a selection of events, filtering based on their name using a regular expression.

In this example the eventListener's HandleEvent handler method will be invoked for any event that begins with 'Cl'.

5.3.4. Using depends - on

For most situations, the fact that an object is a dependency of another is

expressed by the fact that one object is set as a property of another. This is typically accomploished with the <ref/> element in XML-based configuration metadata. For the relatively infrequent situations where dependencies between objects are less direct (for example, when a static initializer in a class needs to be triggered) the 'depends - on' attribute may be used to explicitly force one or more objects to be initialized before the object using this element is initialized. Find below an example of using the 'depends - on' attribute to express a dependency on a single object...

```
<object id="objectOne" type="Examples.ExampleObject, Example</pre>
  property name="manager" ref="manager"/>
</object>
<object id="manager" type="Examples.ManagerObject, Examplest</pre>
```

If you need to express a dependency on multiple objects, you can supply a list of object names as the value of the 'depends-on' attribute, with commas, whitespace and semicolons all valid delimiters, like so:

```
<object id="objectOne" type="Examples.ExampleObject, Example</pre>
  property name="manager" ref="manager" />
</object>
<object id="manager" type="Examples.ManagerObject, Examplest</pre>
<object id="accountDao" type="Examples.AdoAccountDao, Exampl</pre>
```



The 'depends - on' attribute and property is used not only to specify an initialization time dependency, but also to specify the corresponding destroy time dependency (in the case of singleton objects only). Dependent objects that are defined in the 'depends-on' attribute will be destroyed after the relevant object itself is destroyed. This thus allows you to control shutdown order too

5.3.5. Lazily-instantiated objects

The default behavior for IApplicationContext implementations is to eagerly pre-instantiate all singleton objects at startup. Pre-instantiation means that an **IApplicationContext** will eagerly create and configure all of its singleton objects as part of its initialization process. Generally this is a good thing, because it means that any errors in the configuration or in the surrounding environment will be discovered immediately (as opposed to possibly hours or even days down the line).

However, there are times when this behavior is not what is wanted. If you do not want a singleton object to be pre-instantiated when using an <code>IApplicationContext</code>, you can selectively control this by marking an object definition as lazy-initialized. A lazily-initialized object indicates to the IoC container whether or not an object instance should be created at startup or when it is first requested.

When configuring objects via XML, this lazy loading is controlled by the 'lazy-init' attribute on the <object/> element; for example:

```
<object id="lazy" type="MyCompany.ExpensiveToCreateObject, N
<object name="not.lazy" type="MyCompany.AnotherObject, MyApp</pre>
```

When the above configuration is consumed by an IApplicationContext, the object named 'lazy' will not be eagerly pre-instantiated when the IApplicationContext is starting up, whereas the 'not.lazy' object will be eagerly pre-instantiated.

One thing to understand about lazy-initialization is that even though an object definition may be marked up as being lazy-initialized, if the lazy-initialized object is the dependency of a singleton object that is not lazy-initialized, when the <code>IApplicationContext</code> is eagerly pre-instantiating the singleton, it will have to satisfy all of the singletons dependencies, one of which will be the lazy-initialized object! So don't be confused if the IoC container creates one of the objects that you have explicitly configured as lazy-initialized at startup; all that means is that the lazy-initialized object is being injected into a non-lazy-initialized singleton object elsewhere.

It is also possible to control lazy-initialization at the container level by using the 'default-lazy-init' attribute on the <objects/> element; for example:

```
<objects default-lazy-init="true">
```

5.3.6. Autowiring collaborators

The Spring container is able to autowire relationships between collaborating objects. This means that it is possible to automatically let Spring resolve collaborators (other objects) for your object by inspecting the contents of the IoC container.. The autowiring functionality has five modes. Autowiring is specified per object and can thus be enabled for some object, while other objects will not be autowired. Using autowiring, it is possible to reduce or eliminate the need to specify properties or constructor arguments, thus saving a significant amount of typing. When using XML-based configuration metadata, the autowire mode for an object definition is specified by using the autowire attribute of the <object/>element. The following values are allowed:

Table 5.3. Autowiring modes

Mode	Explanation		
no	No autowiring at all. This is the default value and you are encouraged not to change this for large applications, since specifying your collaborators explicitly gives you a feeling for what you're actually doing (always a bonus) and is a great way of somewhat documenting the structure of your system.		
byName	This option will inspect the objects within the container, and look for an object named exactly the same as the property which needs to be autowired. For example, if you have an object definition that is set to autowire by name, and it contains a Master property, Spring.NET will look for an object definition named Master, and use it as the value of the Master property on your object definition.		
byType	This option gives you the ability to resolve collaborators by type instead of by name. Supposing you have an <code>IObjectDefinition</code> with a collaborator typed <code>SqlConnection</code> , Spring.NET will search the entire object factory for an object definition of type <code>SqlConnection</code> and use it as the collaborator. If <code>0</code> (zero) or more than <code>1</code> (one) object definitions of the desired type exist in the container, a failure will be reported and you won't be able to use autowiring for that specific object.		
constructor	This is analogous to <i>byType</i> , but applies to constructor arguments. If there isn't exactly one object of the constructor argument type in the object factory, a fatal error is raised.		
autodetect	Chooses <i>constructor</i> or <i>byType</i> through introspection of the object class. If a default constructor is found, <i>byType</i> gets applied.		

Note that explicit dependencies in property and constructor-arg settings always override autowiring. Please also note that it is not currently possible to autowire so-called simple properties such as primitives, Strings, and Types (and arrays of such simple properties). (This is by-design and should be considered a feature.) When using either the byType or constructor autowiring mode, it is possible to wire arrays and typed-collections. In such cases all autowire candidates within the container that match the expected type will be provided to satisfy the dependency. Strongly-typed IDictionaries can even be autowired if the expected key type is string. An autowired IDictionary values will consist of all object instances that match the expected type, and the IDictionary's keys will contain the corresponding object names.

Autowire behavior can be combined with dependency checking, which will be performed after all autowiring has been completed. It is important to understand the various advantages and disadvantages of autowiring. Some advantages of autowiring include:

- Autowiring can significantly reduce the volume of configuration required. However, mechanisms such as the use of a object template (discussed elsewhere in this chapter) are also valuable in this regard.
- Autowiring can cause configuration to keep itself up to date as your objects evolve. For example, if you need to add an additional dependency to a class, that dependency can be satisfied automatically without the need to modify configuration. Thus there may be a strong case for autowiring during development, without ruling out the option of switching to explicit wiring when the code base becomes more stable.

Some disadvantages of autowiring:

- Autowiring is more magical than explicit wiring. Although, as noted in the above table, Spring is careful to avoid guessing in case of ambiguity which might have unexpected results, the relationships between your Springmanaged objects are no longer documented explicitly.
- Wiring information may not be available to tools that may generate documentation from a Spring container.

Another issue to consider when autowiring by type is that multiple object

definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or IDictionary, this is not necessarily a problem. However for dependencies that expect a single value, this ambiguity will not be arbitrarily resolved. Instead, if no unique object definition is available, an Exception will be thrown.

When deciding whether to use autowiring, there is no wrong or right answer in all cases. A degree of consistency across a project is best though; for example, if autowiring is not used in general, it might be confusing to developers to use it just to wire one or two object definitions.

5.3.7. Checking for dependencies

Spring.NET has the ability to try to check for the existence of unresolved dependencies of an object deployed into the container. These are properties of the object, which do not have actual values set for them in the object definition, or alternately provided automatically by the autowiring feature.

This feature is sometimes useful when you want to ensure that all properties (or all properties of a certain type) are set on an object. Of course, in many cases an object class will have default values for many properties, or some properties do not apply to all usage scenarios, so this feature is of limited use. Dependency checking can also be enabled and disabled per object, just as with the autowiring functionality. The default dependency checking mode is to *not* check dependencies. Dependency checking can be handled in several different modes. In XML-based configuration, this is specified via the <code>dependency-check</code> attribute in an object definition, which may have the following values.

Mode	Explanation	
none	No dependency checking. Properties of the object which have no value specified for them are simply not set.	
simple	Dependency checking is done for primitive types and collections (this means everything except collaborators).	
object	Dependency checking is done for collaborators only.	
all	Dependency checking is done for collaborators, primitive types and collections.	

5.3.8. Method Injection

For most users, the majority of the objects in the container will be singletons. When a singleton object needs to collaborate with (use) another singleton object, or a non-singleton object needs to collaborate with another non-singleton object, the typical and common approach of handling this dependency (by defining one object to be a property of the other) is quite adequate. There is however a problem when the object lifecycles are different. Consider a singleton object A which needs to use a non-singleton (prototype) object B, perhaps on each method invocation on A. The container will only create the singleton object A once, and thus only get the opportunity to set its properties once. There is no opportunity for the container to provide object A with a new instance of object B every time one is needed.

One solution to this problem is to forego some inversion of control. Object A can be <u>made aware of the container</u> by implementing the IObjectFactoryAware interface, and <u>use programmatic means</u> to ask the container via a GetObject("B") call for (a typically new) object B every time it needs it. Find below an admittedly somewhat contrived example of this approach

```
using System.Collections;
using Spring.Objects.Factory;

namespace Fiona.Apple
{
   public class CommandManager : IObjectFactoryAware
   {
      private IObjectFactory objectFactory;

      public object Process(IDictionary commandState)
      {
            // grab a new instance of the appropriate Command Command command = CreateCommand();
            // set the state on the (hopefully brand new) Command.State = commandState;
            return command.Execute();
      }
}
```

```
// the Command returned here could be an implementat
protected Command CreateCommand()
{
    return (Command) objectFactory.GetObject("commar
}

public IObjectFactory ObjectFactory
{
    set { objectFactory = value; }
}
}
```

The above example is generally not a desirable solution since the business code is then aware of and coupled to the Spring Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, allows this use case to be handled in a clean fashion.

5.3.8.1. Lookup Method Injection

Lookup method injection refers to the ability of the container to override abstract or concrete methods on *container managed objects*, to return the result of looking up another named object in the container. The lookup will typically be of a prototype object as in the scenario described above. The Spring framework implements this method injection by a dynamically generating a subclass overriding the method using the classes in the System.Reflection.Emit namespace. You can read more about the motivation for Method Injection in this blog entry.

So if you look at the code from the previous code snipped (the CommandManager class), the Spring container is going to dynamically override the implementation of the CreateCommand() method. Your CommandManager class is not going to have any Spring dependencies, as can be seen in this reworked example below:

```
using System.Collections;
namespace Fiona.Apple
{
   public abstract class CommandManager
```

```
public object Process(IDictionary commandState)
{
    Command command = CreateCommand();
    command.State = commandState;
    return command.Execute();
}

// okay... but where is the implementation of this man protected abstract Command CreateCommand();
}
```

In the client class containing the method to be injected (the CommandManager in this case) the method definition must observe the following form:

```
<public|protected> [abstract] <return-type> TheMethodName(no
```

If the method is abstract, the dynamically-generated subclass will implement the method. Otherwise, the dynamically-generated subclass will override the concrete method defined in the original class. Let's look at an example:

The object identified as **CommandManager** will call its own method **CreateCommand** whenever it needs a new instance of the **Command** object. It is important to note that the person deploying the objects must be careful to deploy the **Command** object as prototype (if that is actually what is needed). If it is deployed as a <u>singleton</u> the same instance of

singleShotHelper will be returned each time!

Note that lookup method injection can be combined with Constructor Injection (supplying optional constructor arguments to the object being constructed), and also with Setter Injection (settings properties on the object being constructed).

5.3.8.2. Arbitrary method replacement

A less commonly useful form of method injection than Lookup Method Injection is the ability to replace arbitrary methods in a managed object with another method implementation. Users may safely skip the rest of this section (which describes this somewhat advanced feature), until this functionality is actually needed.

In an XmlObjectFactory, the replaced-method element may be used to replace an existing method implementation with another. Consider the following class, with a method ComputeValue, which we want to override:

```
public class MyValueCalculator {
  public virtual string ComputeValue(string input) {
    // ... some real code
  }
  // ... some other methods
}
```

A class implementing the Spring.Objects.Factory.Support.IMethodReplacer interface is needed to provide the new (injected) method definition.

```
// compute...
return result;
}
```

The object definition to deploy the original class and specify the method override would look like this:

One or more contained arg-type elements within the replaced-method element may be used to indicate the method signature of the method being overridden. Note that the signature for the arguments is actually only needed in the case that the method is actually overloaded and there are multiple variants within the class. For convenience, the type string for an argument may be a substring of the fully qualified type name. For example, all the following would match System.String.

```
System.String
String
Str
```

Since the number of arguments is often enough to distinguish between each possible choice, this shortcut can save a lot of typing, by just using the shortest string which will match an argument.

5.3.9. Setting a reference using the members of other objects and classes.

This section details those configuration scenarios that involve the setting of properties and constructor arguments using the members of other objects and classes. This kind of scenario is quite common, especially when dealing with

legacy classes that you cannot (or won't) change to accommodate some of Spring.NET's conventions... consider the case of a class that has a constructor argument that can only be calculated by going to say, a database. The MethodInvokingFactoryObject handles exactly this scenario ... it will allow you to inject the result of an arbitrary method invocation into a constructor (as an argument) or as the value of a property setter. Similarly, PropertyRetrievingFactoryObject allow you to retrieve values from another object's property or field value. These classes implement the IFactoryObject interface which indicates to Spring.NET that this object is itself a factory and the factories product, not the factory itself, is what will be associated with the object id. Factory objects are discussed further in Section 5.9.3, "Customizing instantiation logic using IFactoryObjects"

5.3.9.1. Setting a reference to the value of property.

The PropertyRetrievingFactoryObject is an IFactoryObject that addresses the scenario of setting one of the properties and / or constructor arguments of an object to the value of a property exposed on another object or class. One can use it to get the value of any public property exposed on either an instance or a class (in the case of a property exposed on a class, the property must obviously be static).

In the case of a property exposed on an instance, the target object that a <code>PropertyRetrievingFactoryObject</code> will evaluate can be either an object instance specified directly inline or a reference to another arbitrary object. In the case of a static property exposed on a class, the target object will be the class (the .NET <code>System.Type</code>) exposing the property.

The result of evaluating the property lookup may then be used in another object definition as a property value or constructor argument. Note that nested properties are supported for both instance and class property lookups. The **IFactoryObject** is discussed more generally in <u>Section 5.9.3</u>, "Customizing instantiation logic using IFactoryObjects".

Here's an example where a property path is used against another object instance. In this case, an inner object definition is used and the property path is nested, i.e. spouse.age.

An example of using a PropertyRetrievingFactoryObject to evaluate a Static property is shown below.

Similarly, an example showing the use of an instance property is shown below.

5.3.9.2. Setting a reference to the value of field.

The FieldRetrievingFactoryObject class addresses much the same area of concern as the PropertyRetrievingFactoryObject described in the previous section. However, as its name might suggest, the FieldRetrievingFactoryObject class is concerned with looking up the value of a public field exposed on either an instance or a class (and similarly, in the case of a field exposed on a class, the field must obviously be static).

The following example demonstrates using a FieldRetrievingFactoryObject to look up the value of a (public, static) field exposed on a class

The example in the next section demonstrates the look up of a (public) field exposed on an object instance.

5.3.9.3. Setting a property or constructor argument to the return value of a method invocation.

The MethodInvokingFactoryObject rounds out the trio of classes that permit the setting of properties and constructor arguments using the members of other objects and classes. Whereas the PropertyRetrievingFactoryObject and FieldRetrievingFactoryObject classes dealt with looking up and returning the value of property or field on an object or class, the MethodInvokingFactoryObject allows one to set a constructor or property to the return value of an arbitrary method invocation,

The MethodInvokingFactoryObject class handles both the case of invoking an (instance) method on another object in the container, and the case of a static method call on an arbitrary class. Additionally, it is sometimes necessary to invoke a method just to perform some sort of initialization.... while the mechanisms for handling object initialization have yet to be introduced (see Section 5.6.1.1, "IlnitializingObject / init-method"), these mechanisms do not permit any arguments to be passed to any initialization method, and are confined to invoking an initialization method on the object that has just been instantiated by the container. The MethodInvokingFactoryObject allows one to invoke pretty much any method on any object (or class in the case of a static method).

The following example (in an XML based IObjectFactory definition) uses the MethodInvokingFactoryObject class to force a call to a static factory method prior to the instantiation of the object...

Note that the definition for the <code>myService</code> object has used the <code>depends-on</code> attribute to refer to the <code>force-init</code> object, which will force the initialization of the <code>force-init</code> object first (and thus the calling of its configured <code>StaticMethod</code> static initializer method, when <code>myService</code> is first initialized. Please note that in order to effect this initialization, the <code>MethodInvokingFactoryObject</code> object <code>must</code> be operating in <code>singleton</code> mode (the default.. see the next paragraph).

Note that since this class is expected to be used primarily for accessing factory methods, this factory defaults to operating in Singleton mode. As such, as soon as all of the properties for a MethodInvokingFactoryObject object have been set, and if the MethodInvokingFactoryObject object is still in Singleton mode, the method will be invoked immediately and the return value cached for later access. The first request by the container for the factory to produce an object will cause the factory to return the cached return value for the current request (and all subsequent requests). The IsSingleton property may be set to false, to cause this factory to invoke the target method each time it is asked for an object (in this case there is obviously no caching of the return value).

A static target method may be specified by setting the targetMethod property to a string representing the static method name, with TargetType specifying the Type that the static method is defined on. Alternatively, a target instance method may be specified, by setting the TargetObject property to the name of another Spring.NET managed object definition (the target object), and the TargetMethod property to the name of the method to call on that target object.

Arguments for the method invocation may be specified in two ways (or even a mixture of both)... the first involves setting the **Arguments** property to the

list of arguments for the method that is to be invoked. Note that the ordering of these arguments is significant... the order of the values passed to the <code>Arguments</code> property must be the same as the order of the arguments defined on the method signature, including the argument <code>Type</code>. This is shown in the example below

The second way involves passing an arguments dictionary to the NamedArguments property... this dictionary maps argument names (Strings) to argument values (any object). The argument names are not casesensitive, and order is (obviously) not significant (since dictionaries by definition do not have an order). This is shown in the example below

```
</property>
</object>
```

The following example shows how use MethodInvokingFactoryObject to call an instance method.

The above example could also have been written using an anonymous inner object definition... if the object on which the method is to be invoked is not going to be used outside of the factory object definition, then this is the preferred idiom because it limits the scope of the object on which the method is to be invoked to the surrounding factory object.

Finally, if you want to use MethodInvokingFactoryObject in conjunction with a method that has a variable length argument list, then please note that the variable arguments need to be passed (and configured) as a *list*. Let us consider the following method definition that uses the params keyword (in C#), and its attendant (XML) configuration...

```
[C#]
public class MyClassFactory
{
    public object CreateObject(Type objectType, params strin
        {
        return ... // implementation elided for clarity...
     }
}
```

5.3.10. Provided IFactoryObject implementations

In addition to PropertyRetrievingFactoryObject, MethodInvokingFactoryObject, and FieldRetrievingFactoryObject Spring.NET comes with other useful implementations of the IFactoryObject interface. These are discussed below.

5.3.10.1. Common logging

The LogFactoryObject is useful when you would like to share a Common.Logging log object across a number of classes instead of creating a logging instance per class or class hierarchy. Information on the Common.Logging project can be found here. In the example shown below the same logging instance, with a logging category name of "DAOLogger", is used in both the SimpleAccountDao and SimpleProductDao data access objects.

5.4. Object Scopes

When you create a object definition what you are actually creating is a recipe for creating actual instances of the class defined by that object definition. The idea that a object definition is a recipe is important, because it means that, just like a class, you can potentially have many object instances created from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular object definition, but also the scope of the objects created from a particular object definition. This approach is very powerful and gives you the flexibility to choose the scope of the objects you create through configuration instead of having to 'bake in' the scope of an object at the .NET class level. Objects can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports exactly five scopes (of which three are available only if you are using a webaware ApplicationContext).

The scopes supported out of the box are listed below:

Table 5.5.	Object	Scopes
-------------------	---------------	---------------

Scope	Description	
singleton	Scopes a single object definition to a single object instance per Spring IoC container.	
prototype	Scopes a single object definition to any number of object instances.	
request	Scopes a single object definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of an object created off the back of a single object definition. Only valid in the context of a web-aware Spring ApplicationContext.	
session	Scopes a single object definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.	
application	Scopes a single object definition to the lifecycle of a web application. Only valid in the context of a web-aware Spring ApplicationContext.	

5.4.1. The singleton scope

When an object is a singleton, only one shared instance of the object will be managed, and all requests for objects with an id or ids matching that object definition will result in that one specific object instance being returned by the Spring container.

To put it another way, when you define an object definition and it is scoped as a singleton, then the Spring IoC container will create exactly one instance of the object defined by that object definition. This single instance will be stored in a cache of such singleton object, and all subsequent requests and references for that named object will result in the cached object being returned.

Please be aware that Spring's concept of a singleton object is quite different from the Singleton pattern as defined in the seminal Gang of Four (GoF) patterns book. The GoF Singleton hard codes the scope of an object such that one and only one instance of a particular class will ever be created per ApplicationDomain. The scope of the Spring singleton is best described as per container and per object. This means that if you define one object for a particular class in a single Spring container, then the Spring container will create one and only one instance of the class defined by that object definition. The singleton scope is the default scope in Spring. To define an object as a singleton in XML, you would write configuration like so:

```
<object id="accountService" type="MyApp.DefaultAccountServic
<!-- the following is equivalent, though redundant (singleto
<object id="accountService" type="MyApp.DefaultAccountService")</pre>
```

5.4.2. The prototype scope

The non-singleton, prototype scope of object deployment results in the creation of a new object instance every time a request for that specific object is made (that is, it is injected into another object or it is requested via a programmatic GetObject() method call on the container). As a rule of thumb, you should use the prototype scope for all objects that are stateful, while the singleton scope should be used for stateless objects.

To define an object as a prototype in XML, you would write configuration like so:

There is one quite important thing to be aware of when deploying an object in the prototype scope, in that the lifecycle of the object changes slightly. Spring does not manage the complete lifecycle of a prototype object: the container instantiates, configures, decorates and otherwise assembles a prototype object, hands it to the client and then has no further knowledge of that prototype instance. This means that while *initialization* lifecycle callback methods will be called on all objects regardless of scope, in the case of prototypes, any configured *destruction* lifecycle callbacks will *not* be called. It is the responsibility of the client code to clean up prototype scoped objects and release any expensive resources that the prototype object(s) are holding onto. (One possible way to get the Spring container to release resources used by singleton-scoped objects is through the use of a custom object post processor which would hold a reference to the objects that need to be cleaned up.)

In some respects, you can think of the Spring containers role when talking about a prototype-scoped object as somewhat of a replacement for the C# 'new' operator. All lifecycle aspects past that point have to be handled by the client. (The lifecycle of a object in the Spring container is further described in the section entitled "Lifecycle callbacks".)

5.4.3. Singleton objects with prototype-object dependencies

When using singleton-scoped objects that have dependencies on objects that are scoped as prototypes, please be aware that dependencies are resolved at instantiation time. This means that if you dependency inject a prototype-scoped object into a singleton-scoped object, a brand new prototype object will be instantiated and then dependency injected into the singleton object... but that is all. That exact same prototype instance will be the sole instance that is ever supplied to the singleton-scoped object, which is fine if that is what you want.

However, sometimes what you actually want is for the singleton-scoped object to be able to acquire a brand new instance of the prototype-scoped object again and again and again at runtime. In that case it is no use just dependency injecting a prototype-scoped object into your singleton object, because as explained above, that only happens once when the Spring container is instantiating the singleton object and resolving and injecting its dependencies. If you are in the scenario

where you need to get a brand new instance of a (prototype) object again and again and again at runtime, you are referred to the section entitled "Method Injection".

5.4.4. The other scopes

The other scopes, namely request, session, and application are for use only in web-based applications. Please refer to the <u>web documentation on object scopes</u> for more information.

5.5. Type conversion

Type converters are responsible for converting objects from one type to another. When using the XML based file to configure the IoC container, string based property values are converted to the target property type. Spring will rely on the support conversion unless standard .NET for type an alternative TypeConverter is registered for a given type. How to register custom TypeConverters will be described shortly. As a reminder, the standard .NET type converter support works by associating a TypeConverter attribute with the class definition by passing the type of the converter as an attribute argument. [3] For example, an abbreviated class definition for the BCL type Font is shown below.

```
[Serializable, TypeConverter(typeof(FontConverter)), ...]
public sealed class Font : MarshalByRefObject, ICloneable, I
{
   // Methods
   ... etc ..
}
```

5.5.1. Type Conversion for Enumerations

The default type converter for enumerations is the System.ComponentModel.EnumConverter class. To specify the value for an enumerated property, simply use the name of the property. For example the TestObject class has a property of the enumerated type FileMode. One of the values for this enumeration is named Create. The following XML fragment shows how to configure this property

5.5.2. Built-in TypeConverters

Spring.NET pre-registers a number of custom TypeConverter instances

(for example, to convert a type expressed as a string into a real System.Type object). Each of those is listed below and they are all located in the Spring.Objects.TypeConverters namespace of the Spring.Core library.

Table 5.6. Built-in TypeConverters

Type	Explanation
RuntimeTypeConverter	Parses strings representing System. Types to a System. Types and the other way around.
FileInfoConverter	Capable of resolving strings to a System.IO.FileInfo object.
StringArrayConverter	Capable of resolving a comma-delimited list of strings string-array and vice versa.
UriConverter	Capable of resolving a string representation of a Uri to actual Uri-object.
CredentialConverter	Capable of resolving a string representation of a crede for Web client authentication into an instance of System.Net.ICredentials
StreamConverter	Capable of resolving Spring IResource Uri (string) to it corresponding InputStream-object.
ResourceConverter	Capable of resolving Spring IResource Uri (string) to a IResource object.
ResourceManagerConverter	Capable of resolving a two part string (resource name, assembly name) to a System.Resources.ResourceMana object.
RgbColorConverter	Capable of resolving a comma separated list of Red, G Blue integer values to a System.Drawing.Color structure.
ExpressionConverter	Capable of resolving a string into an instance of an objuthat implements the <code>IExpression</code> interface.
NameValueConverter	Capable of resolving an XML formatted string to a Specialized.NameValueCollecti
RegexConverter	Capable of resolving a string into an instance of Reg
RegistryKeyConverter	Capable of resolving a string into a Microsoft.Win32.RegistryKey obj

Spring.NET uses the standard .NET mechanisms for the resolution of System.Types, including, but not limited to checking any configuration files associated with your application, checking the Global Assembly Cache (GAC), and assembly probing.

5.5.3. Custom Type Conversion

There are a few ways to register custom type converters. The fundamental for storage area in Spring custom type converters is the TypeConverterRegistry class. The most convenient way if using an XML implementation of **IObjectFactory** based IApplicationContext is to use the custom configuration section handler TypeConverterSectionHandler This is demonstrated in section Section 5.11, "Configuration of IApplicationContext"

An alternate approach, present for legacy reasons in the port of Spring.NET from the Java code base, is to use the object factory post-processor Spring.Objects.Factory.Config.CustomConverterCo This is described in the next section.

If you are constructing your IoC container Programatically then you should use the RegisterCustomConverter(Type requiredType, TypeConverter converter) method of the ConfigurableObjectFactory interface.

5.5.3.1. Using CustomConverterConfigurer

This section shows in detail how to define a custom type converter that does not use the .NET **TypeConverter** attribute. The type converter class is standalone and inherits from the **TypeConverter** class. It uses the legacy factory post-processor approach.

Consider a user class *ExoticType*, and another class *DependsOnExoticType* which needs ExoticType set as a property:

```
public class ExoticType
{
```

```
private string name;

public ExoticType(string name)
{
    this.name = name;
}

public string Name
{
    get { return this.name; }
}
```

and

```
public class DependsOnExoticType
{
   public DependsOnExoticType() {}

   private ExoticType exoticType;

   public ExoticType ExoticType
   {
      get { return this.exoticType; }
      set { this.exoticType = value; }
   }

   public override string ToString()
   {
      return exoticType.Name;
   }
}
```

When things are properly set up, we want to be able to assign the type property as a string, which a TypeConverter will convert into a real ExoticType object behind the scenes:

The TypeConverter looks like this:

```
public class ExoticTypeConverter : TypeConverter
        public ExoticTypeConverter()
        public override bool CanConvertFrom (
                ITypeDescriptorContext context,
                Type sourceType)
        {
                if (sourceType == typeof (string))
                {
                         return true;
                return base.CanConvertFrom (context, sourceT
        }
        public override object ConvertFrom (
                ITypeDescriptorContext context,
                CultureInfo culture, object value)
        {
                string s = value as string;
                if (s != null)
                         return new ExoticType(s.ToUpper());
                return base.ConvertFrom (context, culture, v
        }
}
```

Finally, we use the CustomConverterConfigurer to register the new TypeConverter with the IApplicationContext, which will then be able to use it as needed:

5.6. Customizing the nature of an object

5.6.1. Lifecycle interfaces

Spring.NET uses several marker interfaces to change the behaviour of your container, Spring.NET object in the namely the specific IInitializingObject standard interface the and System.IDisposable interfaces. **Implementing** either of the aforementioned interfaces will in the result container the AfterPropertiesSet() method for the former the and Dispose() method for the latter, thus allowing you to do things upon the initialization and destruction of your objects.

Spring.NET implementations of the Internally, uses **IObjectPostProcessor** interface to process any marker interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring.NET doesn't offer out-of-the-box, you can implement an IObjectPostProcessor yourself. More information about this can be found in Section 5.9.1, "Customizing objects with IObjectPostProcessors".

All the different lifecycle marker interfaces are described below.

5.6.1.1. IlnitializingObject / init-method

Spring.Objects.Factory.IInitializingObject The interface gives you the ability to perform initialization work after all the an object are set by the necessary properties on container. The IInitializingObject interface specifies exactly one method:

• void AfterPropertiesSet(): called after all properties have been set by the container. This method enables you to do checking to see if all necessary properties have been set correctly, or to perform further initialization work. You can throw any Exception to indicate misconfiguration, initialization failures, etc.



Generally, the use of the IInitializingObject can be avoided. The Spring. Core library provides support for a generic init-method, given to the object definition in the object

configuration store (be it XML, or a database, etc).

```
<object id="exampleInitObject" type="Examples.ExampleObject"</pre>
[C#]
public class ExampleObject
    public void Init()
    {
        // do some initialization work
    }
}
```

Is exactly the same as...

```
<object id="exampleInitObject" type="Examples.AnotherExample"</pre>
[C#]
public class AnotherExampleObject : IInitializingObject
    public void AfterPropertiesSet()
        // do some initialization work
}
```

but does not couple the code to Spring.NET.



When deploying an object in **prototype** mode, the lifecycle of the object changes slightly. By definition, Spring.NET cannot manage the complete lifecycle of a non-singleton / prototype object, since after it is created, it is given to the client and the container no longer keeps a reference to the object. You can think of Spring.NET's role when talking about a non-singleton (prototype) object as a replacement for the **new** operator. Any lifecycle aspects past that point have to be handled by the client.

5.6.1.2. IDisposable / destroy-method

The System. IDisposable interface provides you with the ability to get

a callback when an IObjectFactory is destroyed. The IDisposable interface specifies exactly one method:

• **void Dispose()**: and is called on destruction of the container. This allows you to release any resources you are keeping in this object (such as database connections). You can throw any Exception here... however, any such Exception will not stop the destruction of the container - it will only get logged.



Note: If you choose you can avoid having your class implement IDisposable since the Spring. Core library provides support for a generic destroy-method, given to the object definition in the object configuration store (be it XML, or a database, etc).

```
<object id="exampleInitObject" type="Examples.ExampleObject"</pre>
[C#]
public class ExampleObject
    public void cleanup()
        // do some destruction work (such as closing any ope
    }
}
```

is exactly the same as:

```
<object id="exampleInitObject" type="Examples.AnotherExample"</pre>
[C#]
public class AnotherExampleObject : IDisposable
    public void Dispose()
    {
        // do some destruction work
    }
}
```

5.6.2. Knowing who you are

5.6.2.1. IObjectFactoryAware

A class which implements the Spring.Objects.Factory.IObjectFactoryAware interface is provided with a reference to the IObjectFactory that created it. The interface specifies one (write-only) property:

• IObjectFactory ObjectFactory: the property that will be set after the initialization methods (AfterPropertiesSet and the init-method).

This allows objects to manipulate the <code>IObjectFactory</code> that created them Programatically, through the <code>IObjectFactory</code> interface, or by casting the reference to a known subclass of this which exposes additional functionality. Primarily this would consist of programmatic retrieval of other objects. While there are cases when this capability is useful, it should generally be avoided, since it couples the code to Spring.NET, and does not follow the Inversion of Control style, where collaborators are provided to objects as properties.

5.6.2.2. IObjectNameAware

The Spring.Objects.Factory.IObjectNameAware interface gives you the ability to let the container set the name of the object definition on the object instance itself. In those cases where your object needs to know what its name is, implement this interface.

• **string ObjectName**: the property that will be set to let the object know what its name is.

5.7. Object definition inheritance

An object definition potentially contains a large amount of configuration information, including container specific information (i.e. initialization method, static factory method name, etc.) and constructor arguments and property values. A child object definition is an object definition that inherits configuration data from a parent definition. It is then able to override some values, or add others, as needed. Using parent and child object definitions can potentially save a lot of typing. Effectively, this is a form of templating.

When working with an <code>IObjectFactory</code> Programatically, child object definitions are represented by the <code>ChildObjectDefinition</code> class. Most users will never work with them on this level, instead configuring object definitions declaratively in something like the <code>XmlObjectFactory</code>. In an <code>XmlObjectFactory</code> object definition, a child object definition is indicated simply by using the parent attribute, specifying the parent object definition as the value of this attribute.

A child object definition will use the object class from the parent definition if none is specified, but can also override it. In the latter case, the child object class must be compatible with the parent, i.e. it must accept the parent's property values.

A child object definition will inherit constructor argument values, property values and method overrides from the parent, with the option to add new values. If init method, destroy method and/or static factory method are specified, they will override the corresponding parent settings.

The remaining settings will always be taken from the child definition: depends on, autowire mode, dependency check,

singleton, lazy init.

In the case where the parent definition does not specify a class...

```
<object id="inheritedTestObjectWithoutClass" abstract="true"</pre>
  property name="name" value="parent"/>
  operty name="age" value="1"/>
</object>
<object id="inheritsWithClass" type="Spring.Objects.Derived]</pre>
  parent="inheritedTestObjectWithoutClass" init-method="Init
  property name="name" value="override"/>
  <!-- age will inherit value of 1 from parent -->
</object>
```

... the parent object cannot be instantiated on its own since it incomplete, and it is also explicitly marked as abstract. When a definition is defined to be abstract like this, it is usable only as a pure template object definition that will serve as a parent definition for child definitions. Trying to use such an abstract parent object on its own (by referring to it as a ref property of another object, or doing an explicit GetObject() with the parent object id), will result in an error. The container's internal PreInstantiateSingletons method will completely ignore object definitions that are considered abstract.



Application contexts (but not simple object factories) will by default pre-instantiate all singletons. Therefore it is important (at least for singleton objects) that if you have a (parent) object definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the abstract attribute to true, otherwise the application context will actually (attempt to) preinstantiate the abstract object.

5.8. Interacting with the container

The Spring container is essentially nothing more than an advanced factory capable of maintaining a registry of different objects and their dependencies. The <code>IObjectFactory</code> enables you to read object definitions and access them using the object factory. When using just the <code>IObjectFactory</code> you would create an instance of one and then read in some object definitions in the XML format as follows:

```
[C#]
IResource input = new FileSystemResource ("objects.xml");
XmlObjectFactory factory = new XmlObjectFactory(input);
```

That is pretty much it. Using GetObject(string) (or the more concise indexer method factory ["string"]) you can retrieve instances of your objects...

```
[C#]
object foo = factory.GetObject ("foo"); // gets the object o
object bar = factory ["bar"]; // same thing, just
```

You'll get a reference to the same object if you defined it as a singleton (the default) or you'll get a new instance each time if you set the **Singleton** property of your object definition to *false*.

```
<object id="exampleObject" type="Examples.ExampleObject, Exa
<object id="anotherObject" type="Examples.ExampleObject, ExampleObject, Exa
```

The client-side view of the IObjectFactory is surprisingly simple. The IObjectFactory interface has only seven methods (and the

aforementioned indexer) for clients to call:

- bool ContainsObject(string): returns true if the IObjectFactory contains an object definition that matches the given name.
- object GetObject(string): returns an instance of the object registered under the given name. Depending on how the object was configured by the IObjectFactory configuration, either a singleton (and thus shared) instance or a newly created object will be returned. An ObjectsException will be thrown when either the object could not found which it'll be (in be case NoSuchObjectDefinitionException), or an exception occurred while instantiated and preparing the object.
- Object this [string]: this is the indexer for the IObjectFactory interface. It functions in all other respects in exactly the same way as the GetObject(string) method. The rest of this documentation will always refer to the GetObject(string) method, but be aware that you can use the indexer anywhere that you can use the GetObject(string) method.
- Object GetObject(string, Type): returns an object, registered under the given name. The object returned will be cast to the given Type. If the object could not be cast, corresponding exceptions will be thrown (ObjectNotOfRequiredTypeException). Furthermore, all rules of the GetObject(string) method apply (see above).
- bool IsSingleton(string): determines whether or not the object definition registered under the given name is a singleton or a prototype. If the object definition corresponding to the given name could not be found, an exception will be thrown (NoSuchObjectDefinitionException)
- string[] GetAliases(string): returns the aliases for the given object name, if any were defined in the IObjectDefinition.
- void ConfigureObject(object target): Injects

dependencies into the supplied target instance. The name of the abstract object definition is the <code>System.Type.FullName</code> of the target instance. This method is typically used when objects are instantiated outside the control of a developer, for example when ASP.NET instantiates web controls and when a WinForms application creates UserControls.

• void ConfigureObject(object target, string name): Offers the same functionality as the previously listed Configure method but uses a named object definition instead of using the type's full name.

A sub-interface of IObjectFactory, IConfigurableObjectFactory adds some convenient methods such as

- void RegisterSingleton(string name, object objectInstance): Register the given existing object as singleton in the object factory under the given object name.
- void RegisterAlias(string name, string theAlias); Given an object name, create an alias.

Check the SDK docs for additional details on IConfigurableObjectFactory methods and properties and the full IObjectFactory class hierarchy.

5.8.1. Obtaining an IFactoryObject, not its product

Sometimes there is a need to ask an IObjectFactory for an actual IFactoryObject instance itself, not the object it produces. This may be done by prepending the object id with & when calling the GetObject method of the IObjectFactory and IApplicationContext interfaces. So for a given IFactoryObject with an id myObject, invoking GetObject("myObject") on the IObjectFactory will return the product of the IFactoryObject, but invoking GetObject("&myObject") will return the IFactoryObject instance itself.

5.9. Container extension points

The IoC component of the Spring Framework has been designed for extension. There is typically no need for an application developer to subclass any of the IObjectFactory **IApplicationContext** or various implementation classes. The Spring IoC container can be infinitely extended by plugging in implementations of special integration interfaces. The next few sections are devoted to detailing all of these various integration interfaces.

Customizing objects 5.9.1. with **IObjectPostProcessors**

The first extension point that will look is the we at Spring.Objects.Factory.Config.IObjectPostProces interface. This interface defines a number of callback methods that you as an application developer can implement in order to provide your own (or override the containers default) instantiation logic, dependency-resolution logic, and so forth. If you want to do some custom logic after the Spring container has finished instantiating, configuring and otherwise initializing an object, you can plug in one or more IObjectPostProcessor implementations.

You can configure multiple IObjectPostProcessors if you wish. You can control the order in which these IObjectPostProcessor execute by setting the 'Order' property (you can only set this property if the IObjectPostProcessor implements the IOrdered interface; if you write your own IObjectPostProcessor you should consider implementing the IOrdered interface too); consult the SDK docs for the IObjectPostProcessor and IOrdered interfaces for more details.



IObjectPostProcessor operate on object instances; that is to say, the Spring IoC container will have instantiated a object instance for you, and then IObjectPostProcessors get a chance to do their stuff. If you want to change the actual object definition (that is the recipe that defines the object), then you rather need to use a IObjectFactoryPostProcessor (described below in the section entitled Customizing configuration metadata with IObjectFactoryPostProcessors.

Also, IObjectPostProcessors are scoped per-container. This is only relevant if you are using container hierarchies. If you define a IObjectPostProcessor in one container, it will only do its stuff on the objects in that container. Objects that are defined in another container will not be post-processed by IObjectPostProcessors in another container, even if both containers are part of the same hierarchy.

The Spring.Objects.Factory.Config.IObjectPostProces interface, which consists of two callback methods shown below.

```
object PostProcessBeforeInitialization(object instance, stri
object PostProcessAfterInitialization(object instance, strir
```

When such a class is registered as a post-processor with the container, for each object instance that is created by the container, (see below for how this registration is effected), for each object instance that is created by the container, the post-processor will get a callback from the container both before any initialization methods (such as the AfterPropertiesSet method of the IInitializingObject interface and any declared init method) are called, and also afterwards. The post-processor is free to do what it wishes with the object, including ignoring the callback completely. An object post-processor will typically check for marker interfaces, or do something such as wrap an object with a proxy. Some Spring.NET AOP infrastructure classes are implemented as object post-processors as they do this proxy-wrapping logic.

Other extensions to the IObjectPostProcessors interface are IInstantiationAwareObjectPostProcessor and IDestructionAwareObjectPostProcessor defined below

```
public interface IInstantiationAwareObjectPostProcessor : IC
{
   object PostProcessBeforeInstantiation(Type objectType,
   bool PostProcessAfterInstantiation(object objectInstanc
   IPropertyValues PostProcessPropertyValues(IPropertyValue)
```

```
public interface IDestructionAwareObjectPostProcessor : IObj
{
    void PostProcessBeforeDestruction (object instance, str
}
```

The PostProcessBeforeInstantiation callback method is called right before the container creates the object. If the object returned by this method is not null then the default instantiation behavior of the container is short circuited. The returned object is the one registered with the container and no other IObjectPostProcessor callbacks will be invoked on it. This mechanism is useful if you would like to expose a proxy to the object instead of the actual target object. The PostProcessAfterInstantiation callback method is called after the object has been instantiated but before Spring performs property population based on explicit properties or autowiring. A return value of false would short circuit the standard Spring based property population. The callback method PostProcessPropertyValues is called after Spring collects all the property values to apply to the object, but before they are applied. This gives you the opportunity to perform additional processing such as making sure that a property is set to a value if it contains a [Required] attribute or to perform attribute based wiring, i.e. adding the attribute [Inject("objectName")] on a property. Both of these features are scheduled to be included in Spring .12.

The IDestructionAwareObjectPostProcessor callback contains a single method, PostProcessBeforeDestruction, which is called before a singleton's destroy method is invoked.

It is important to know that the <code>IObjectFactory</code> treats object post-processors slightly differently than the <code>IApplicationContext</code>. An <code>IApplicationContext</code> will automatically detect any objects which are deployed into it that implement the <code>IObjectPostProcessor</code> interface, and register them as post-processors, to be then called appropriately by the factory on object creation. Nothing else needs to be done other than deploying the post-processor in a similar fashion to any other object. On the other hand, when using plain <code>IObjectFactories</code>, object post-processors have to manually be explicitly registered, with a code sequence such as...

```
ConfigurableObjectFactory factory = new ....; // create an
... // now register some objects
// now register any needed IObjectPostProcessors
MyObjectPostProcessor pp = new MyObjectPostProcessor();
factory.AddObjectPostProcessor(pp);
// now start using the factory
. . .
```

This explicit registration step is not convenient, and this is one of the reasons why the various IApplicationContext implementations are preferred above plain IObjectFactory implementations in the vast majority of Spring-backed applications, especially when using IObjectPostProcessors.



IObjectPostProcessors and AOP auto-proxying Classes that implement the IObjectPostProcessor interface are special, and so they are treated differently by the container. All IObjectPostProcessors and their directly referenced object will be instantiated on startup, as part of the special startup phase of the IApplicationContext, then all those IObjectPostProcessors will be registered in a sorted fashion - and applied to all further objects. Since AOP auto-proxying is implemented as a IObjectPostProcessor itself, no IObjectPostProcessors or directly referenced objects are eligible for auto-proxying (and thus will not have aspects 'woven' into them). For any such object, you should see an info log message: "Object 'foo' is not eligible for getting processed by all IObjectPostProcessors (for example: not eligible for auto-proxying)".

5.9.1.1. Example: Hello World, IObjectPostProcessor-style

This first example is hardly compelling, but serves to illustrate basic usage. All we are going to do is code a custom IObjectPostProcessor implementation that simply invokes the ToString() method of each object as it is created by the container and prints the resulting string to the system console. Yes, it is not hugely useful, but serves to get the basic concepts across before we move into the second example which is actually useful. The basis of the example is the MovieFinder quickstart that is included with the Spring.NET distribution.

Find below the custom IObjectPostProcessor implementation class definition

And the following configuration

Notice how the TracingObjectPostProcessor is simply defined; it doesn't even have a name, and because it is a object it can be dependency injected just like any other object.

Find below a small driver script to exercise the above code and configuration;

The output of executing the above program will be:

5.9.1.2. Example: the RequiredAttributeObjectPostProcessor

Using callback interfaces or annotations in conjunction with a custom IObjectPostProcessor implementation is a common means of extending the Spring IoC container. The [Required] attribute in the Spring.Objects.Factory.Attributes namespace can be used to mark a property as being 'required-to-be-set' (i.e. an setter property with this attribute applied must be configured to be dependency injected with a value), else an ObjectInitializationException will be thrown by the container at runtime.

The best way to illustrate the usage of this attribute is with an example.

```
public class MovieLister
{
    // the MovieLister has a dependency on the MovieFinder
    private IMovieFinder _movieFinder;

    // a setter property so that the Spring container can 'inj
    [Required]
    public IMovieFinder MovieFinder
    {
        set { _movieFinder = value; }
    }

    // business logic that actually 'uses' the injected MovieFinder
}
```

Hopefully the above class definition reads easy on the eye. Any and all **IObjectDefinitions** for the **MovieLister** class must be provided with a value.

Let's look at an example of some XML configuration that will not pass validation.

At runtime the following message will be generated by the Spring container

```
Error creating context 'spring.root': Property 'MovieFinder'
```

There is one last little piece of Spring configuration that is required to actually 'switch on' this behavior. Simply annotating the 'setter' properties of your classes is not enough to get this behavior. You need to enable a component that is aware of the [Required] attribute and that can process it appropriately.

```
This component is the RequiredAttributeObjectPostProcessor class. This is a
```

special IObjectPostProcessor implementation that is [Required]-aware and actually provides the 'blow up if this required property has not been set' logic. It is very easy to configure; simply drop the following object definition into your Spring XML configuration.

```
<object type="Spring.Objects.Factory.Attributes.RequiredAttr</pre>
```

Finally, configure of one can an instance RequiredAttributeObjectPostProcessor class to look for another Attribute type. This is great if you already have your own [Required]-style attribute. Simply plug it into the definition of a RequiredAttributeObjectPostProcessor and you are good to go. By way of an example, let's suppose you (or your organization / team) defined an attribute called [Mandatory]. You make can RequiredAttributeObjectPostProcessor instance [Mandatory]-aware like so:

<object type="Spring.Objects.Factory.Attributes.RequiredAttr
 <pre>cobject type="Spring.Objects.Factory.Attributes.RequiredAttributeType" value="MyApp.AttributeType" value="MyAp

5.9.2. Customizing configuration metadata with ObjectFactoryPostProcessors

will The next extension point that we look the Spring.Objects.Factory.Config.IObjectFactoryPos The semantics of this interface similar are to IObjectPostProcessor, difference. with major one IObjectFactoryPostProcessors operate on; that is to say, the Spring IoC container will allow IObjectFactoryPostProcessors to read the configuration metadata and potentially change it before the container has actually instantiated any other objects. By implementing this interface, you will receive a callback after the all the object definitions have been loaded into the IoC container but before they have been instantiated. The signature of the interface is shown below

```
public interface IObjectFactoryPostProcessor
  void PostProcessObjectFactory (IConfigurableListableObject
}
```

You can configure multiple IObjectFactoryPostProcessors if You control order can the vou IObjectFactoryPostProcessors execute by setting the 'Order' this if property (you can only set property IObjectFactoryPostProcessors implements the IOrdered interface; if you write your own IObjectFactoryPostProcessors you should consider implementing the IOrdered interface too); consult the IObjectFactoryPostProcessors the IOrdered interfaces for more details.



If you want to change the actual object instances (the objects that are created from the configuration metadata), then you rather need to use a IObjectObjectPostProcessor (described above in the section entitled <u>Customizing objects with IObjectPostProcessors</u>.

Also, IObjectFactoryPostProcessors are scoped percontainer. This is only relevant if you are using container hierarchies. If you define a IObjectFactoryPostProcessors in one container, it will only do its stuff on the object definitions in that container. Object definitions in another container will not be postprocessed by IObjectFactoryPostProcessors in another container, even if both containers are part of the same hierarchy.

An object factory post-processor is executed manually (in the case of a IObjectFactory) or automatically (in the case of an IApplicationContext) to apply changes of some sort to the configuration metadata that defines a container. Spring.NET includes a number of pre-existing object factory post-PropertyResourceConfigurer as PropertyPlaceHolderConfigurer, both described below and ObjectNameAutoProxyCreator, which is very useful for wrapping other objects transactionally or with any other kind of proxy, as described later in this manual.

IObjectFactory, the process In of applying an IObjectFactoryPostProcessor is manual, and will be similar to

```
XmlObjectFactory factory = new XmlObjectFactory(new FileSyst
// create placeholderconfigurer to bring in some property
// values from a Properties file
PropertyPlaceholderConfigurer cfg = new Proper
cfg.setLocation(new FileSystemResource("ado.properties"));
// now actually do the replacement
cfg.PostProcessObjectFactory(factory);
```

This explicit registration step is not convenient, and this is one of the reasons why the various IApplicationContext implementations are preferred above plain IObjectFactory implementations in the vast majority of Spring-backed applications, especially when using IObjectFactoryPostProcessors.

An IApplicationContext will detect any objects which are deployed into it that implement the ObjectFactoryPostProcessor interface, and automatically use them as object factory post-processors, at the appropriate time. Nothing else needs to be done other than deploying these post-processor in a similar fashion to any other object.



Just as in the case of IObjectPostProcessors, you typically don't want to have IObjectFactoryPostProcessors marked as being lazily-initialized. If they are marked as such, then the Spring container will never instantiate them, and thus they won't get a chance to apply their custom logic. If you are using the 'default-lazyinit' attribute on the declaration of your <objects/> element, be sure to mark your various IObjectFactoryPostProcessor object definitions with 'lazy-init="false".

Example: 5.9.2.1. **PropertyPlaceholderConfigurer**

The

The PropertyPlaceholderConfigurer is an excellent solution when you want to externalize a few properties from a file containing object definitions. This is useful to allow the person deploying an application to customize environment specific properties (for example database configuration strings, usernames, and passwords), without the complexity or risk of modifying the main XML definition file or files for the container.

Variable substitution is performed on simple property values, lists, dictionaries, sets, constructor values, object type name, and object names in runtime object references. Furthermore, placeholder values can also cross-reference other placeholders.

Note that IApplicationContexts are able to automatically recognize deployed objects them and apply in that implement IObjectFactoryPostProcessor interface. This means that as described here, applying a PropertyPlaceholderConfigurer is much more convenient when using an IApplicationContext. For this reason, it is recommended that users wishing to use this or other object factory IApplicationContext instead postprocessors use an an IObjectFactory.

In the example below a data access object needs to be configured with a database connection and also a value for the maximum number of results to return in a query. Instead of hard coding the values into the main Spring.NET configuration file we use place holders, in the NAnt style of ${\text NAnt style}$ and obtain their values from NameValueSections in the standard .NET application configuration file. The Spring.NET configuration file looks like:

Notice the presence of two NameValueSections in the configuration file. These name value pairs will be referred to in the Spring.NET configuration file. In this example we are using an embedded assembly resource for the location of the Spring.NET configuration file so as to reduce the chance of accidental tampering in deployment. This Spring.NET configuration file is shown below.

```
<objects xmlns="http://www.springframework.net"</pre>
                                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">xmlns:xsi="http://www.w3.org/2001/XMLSchema-inst
                                   xsi:schemaLocation="http://www.springframework.net
                                   http://www.springframework.net/xsd/spring-objects.x
                <object name="productDao" type="DaoApp.SimpleProductDao,</pre>
                       cproperty name="maxResults" value="${maxResults}"/>
                       connection" ref="myConnection"/>
                </object>
                <object name="myConnection" type="System.Data.Odbc.Odbc0</pre>
                       connectionstring" value="${connection.
                </object>
                <object name="appConfigPropertyHolder"</pre>
                                                type="Spring.Objects.Factory.Config.PropertyPlace
                                configSections">
                                        <value>DaoConfiguration,DatabaseConfiguration</val
                                </property>
```

</object> </objects>

The values of \${maxResults} and \${connection.string} key names used in the two NameValueSectionHandlers DatabaseConfiguration. DaoConfiguration and PropertyPlaceholderConfigurer refers to these two sections via a comma delimited list of section names in the configSections property. If you are using section groups, prefix the section group name, for example myConfigSection/DaoConfiguraiton.

The PropertyPlaceholderConfigurer class also retrieving name value pairs from other IResource locations. These can be specified using the Location and Locations properties of the PropertyPlaceHolderConfigurer class.

If there are properties with the same name in different resource locations the default behavior is that the last property processed overrides the previous values. This is behavior is controlled by the LastLocationOverrides property. True enables overriding while false will append the values as one would normally expect using NameValueCollection.Add.



In an ASP.NET environment you must specify the full, four-part name of the ass NameValueFileSectionHandler

> <section name="hibernateConfiguration"</pre> type="System.Configuration.NameValueFileSection| Version=1.0.3300.0, Culture=neutral, PublicKeyTo

5.9.2.1.1. Type, Ref, and Expression substitution

The PropertyPlaceholderConfigurer can be used to substitute type names, which is sometimes useful when you have to pick a particular implementation class at runtime. For example:

```
<object id="MyMovieFinder" type="${custom.moviefinder.type}'</pre>
```

If the class is unable to be resolved at runtime to a valid type, resolution of the object will fail once it is about to be created (which is during the PreInstantiateSingletons() phase of an ApplicationContext for a non-lazy-init object.)

Similarly you can replace 'ref' and 'expression' metadata, as shown below

5.9.2.1.2. Replacement with Environment Variables

You may also use the value environment variables to replace property placeholders. The use of environment variables is controlled via the property EnvironmentVariableMode. This property is an enumeration of the type EnvironmentVariablesMode and has three values, Never, Fallback, and Override. Fallback is the default value and will resolve a property placeholder if it was not already done so via a value from a resource location. Override will apply environment variables before applying values defined from a resource location. **Never** will, quite appropriately, disable environment variable substitution. An example of how PropertyPlaceholderConfigurer XML is modified to enable override usage is shown below

PropertyOverrideConfigurer

The PropertyOverrideConfigurer, another object factory postprocessor, is similar to the PropertyPlaceholderConfigurer, but in contrast to the latter, the original definitions can have default values or no values at all for object properties. If an overriding configuration file does not have an entry for a certain object property, the default context definition is used.

Note that the object factory definition is *not* aware of being overridden, so it is not immediately obvious when looking at the XML definition file that the override configurer is being used. In case that there are multiple PropertyOverrideConfigurer instances that define different values for the same object property, the last one will win (due to the overriding mechanism).

The example usage is similar to when using PropertyPlaceHolderConfigurer except that the key name refers to the name given to the object in the Spring.NET configuration file and is suffixed via 'dot' notation with the name of the property For example, if the application configuration file is

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.0"</pre>
    </sectionGroup>
    <section name="DaoConfigurationOverride" type="System.Co"</pre>
  </configSections>
  <DaoConfigurationOverride>
    <add key="productDao.maxResults" value="1000"/>
  </DaoConfigurationOverride>
  <spring>
    <context>
      <resource uri="assembly://DaoApp/DaoApp/objects.xml"/>
    </context>
  </spring>
</configuration>
```

Then the value of 1000 will be used to overlay the value of 2000 set in the Spring.NET configuration file shown below

```
<objects xmlns="http://www.springframework.net"</pre>
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
        xsi:schemaLocation="http://www.springframework.net h
    <object name="productDao" type="PropPlayApp.SimpleProduct")</pre>
      property name="maxResults" value="2000"/>
      cproperty name="dbConnection" ref="myConnection"/>
      cproperty name="log" ref="daoLog"/>
    </object>
    <object name="daoLog" type="Spring.Objects.Factory.Confi</pre>
      roperty name="logName" value="DAOLogger"/>
    </object>
    <object name="myConnection" type="System.Data.Odbc.Odbc(</pre>
      cproperty name="connectionstring">
        <value>dsn=MyDSN;uid=sa;pwd=myPassword;</value>
      </property>
    </object>
    <object name="appConfigPropertyOverride" type="Spring.Ot</pre>
      configSections">
        <value>DaoConfigurationOverride</value>
      </property>
    </object>
</objects>
```

5.9.2.3. IVariableSource

The IVariableSource is the base interface for providing the ability to get the value of property placeholders (name-value) pairs from a variety of sources. Out of the box, Spring.NET supports a number of variable sources that allow users to obtain variable values from .NET config files, java-style property files, environment variables, command line arguments and the registry and the new connection strings configuration section in .NET 2.0. The list of implementing classes is listed below. Please refer to the SDK documentation for more

information.

- ConfigSectionVariableSource
- PropertyFileVariableSource
- EnvironmentVariableSource
- CommandLineArgsVariableSource
- RegistryVariableSource
- SpecialFolderVariableSource
- ConnectionStringsVariableSource

You use this by defining an instance of Spring.Objects.Factory.Config.VariablePlacehold in your configuration and set the property VariableSource to a single IVariableSource instance or the list property VariableSources to a list of IVariableSource instances. In the case of the same property defined in multiple IVariableSource implementations, the first one in the list that contains the property value will be used.

The IVariableSource interface is shown below

```
public interface IVariableSource
{
   string ResolveVariable(string name);
}
```

This is a simple contract to implement if you should decide to create your own

custom implemention. Look at the source code of the current implementations for some inspiration if you go that route. To register your own custom implementation, simply configure VariablePlaceholderConfigurer to refer to your class.

5.9.3. Customizing instantiation logic using **IFactoryObjects**

The Spring.Objects.Factory.IFactoryObject interface is to be implemented by objects that are themselves factories.

The IFactoryObject interface is a point of pluggability into the Spring IoC containers instantiation logic. If you have some complex initialization code that is better expressed in C# as opposed to a (potentially) verbose amount of XML, you can create your own IFactoryObject, write the complex initialization inside that class, and then plug your custom IFactoryObject into the container.

The **IFactoryObject** interface provides one method and two (read-only) properties:

- **object GetObject()**: has to return an instance of the object this factory creates. The instance can possibly be shared (depending on whether this factory provides singletons or prototypes).
- bool IsSingleton: has to return true if this IFactoryObject returns singletons, false otherwise.
- Type ObjectType: has to return either the object type returned by the GetObject() method or null if the type isn't known in advance.

IFactoryObject

The IFactoryObject concept and interface is used in a number of places within the Spring Framework. Some examples of its use is described in Section 5.3.9, "Setting a reference using the members of other objects and classes." for the PropertyRetrievingFactoryObject and FieldRetrievingFactoryObject. An additional use of creating an custom IFactoryObject implementation is to retrieve an object from an embedded resource file and use it to set another objects dependency. An example of this is provided here.

Finally, there is sometimes a need to ask a container for an actual IFactoryObject instance itself, not the object it produces. This may be achieved by prepending the object id with '&' (sans quotes) when calling the GetObject method of the IObjectFactory (including IApplicationContext). So for a given IFactoryObject with an id of 'myObject', invoking GetObject("myObject") on the container will return the product of the IFactoryObject, but invoking GetObject("&myObject") will return the IFactoryObject instance itself.

5.9.3.1. IConfigurableFactoryObject

The

Spring.Objects.Factory.IConfigurableFactoryObje interface inherits from IFactoryObject interface and adds the following property.

• IObjectDefinition ProductTemplate : Gets the template object definition that should be used to configure the instance of the object managed by this factory.

IConfigurableFactoryObject implementions you already have examples of in <u>Section 27.3</u>, "Client-side" are WebServiceProxyFactory.

5.10. The IApplicationContext

While the Spring.Objects namespace provides basic functionality for managing and manipulating objects, often in a programmatic way, the Spring.Context namespace introduces the IApplicationContext interface, which enhances the functionality provided by the IObjectFactory interface in a more framework-oriented style. Many users will use ApplicationContext in a completely declarative fashion, not even having to create it manually, but instead relying on support classes such as the .NET configuration section handlers such as ContextHandler and WebContextHandler together to declaratively define the ApplicationContext and retrieve it though a ContextRegistry. (Of course it is still possible to create an IApplicationContext Programatically).

The basis for the context module is the IApplicationContext interface, located in the Spring.Context namespace. Deriving from the IObjectFactory interface, it provides all the functionality of the IObjectFactory. To be able to work in a more framework-oriented fashion, using layering and hierarchical contexts, the Spring.Context namespace also provides the following functionality

- Loading of multiple (hierarchical) contexts, allowing some of them to be focused and used on one particular layer, for example the web layer of an application.
- Access to localized resources at the application level by implementing IMessageSource.
- *Uniform access to resources* that can be read in as an InputStream, such as URLs and files by implementing IResourceLoader
- Loosely Coupled Event Propagation. Publishers and subscribers of events do not have to be directly aware of each other as they register their interest indirectly through the application context.

5.10.1. IObjectFactory or IApplicationContext?

Short version: use an *IApplicationContext* unless you have a really good reason for not doing so. For those of you that are looking for slightly more

depth as to the 'but why' of the above recommendation, keep reading.

As the IApplicationContext includes all the functionality the object factory via its inheritance of the IObjectFactory interface, it is generally recommended to be used over the IObjectFactory except for a few limited situations where memory consumption might be critical. This may become more important if the .NET Compact Framework is supported. The history of IObjectFactory comes from the Spring Java framework, where the use of Spring in Applets was a concern to reduce memory consumption. However, for most 'typical' enterprise applications and systems, the IApplicationContext is what you will want to use. Spring generally makes heavy use of the IObjectPostProcessor extension point (to effect proxying and suchlike), and if you are using just a plain IObjectFactory then a fair amount of support such as transactions and AOP will not take effect (at least not without some extra steps on your part), which could be confusing because nothing will actually be wrong with the configuration.

Find below a feature matrix that lists what features are provided by the IObjectFactory and IApplicationContext interfaces (and attendant implementations). The following sections describe functionality that IApplicationContext adds to the basic IObjectFactory capabilities in a lot more depth than the said feature matrix.)

Tahl	<u> </u>	7	Feature	Matrix
Tanı	CJ.	. / •	reature	Manix

Feature	IO bjectFactory	IApplicationCont
Object instantiation/wiring	Yes	Yes
Automatic IObjectPostProcessor registration	No	Yes
Automatic IObjectFactoryPostProcessor registration	No	Yes
Convenient IMessageSource access	No	Yes
ApplicationEvent publication	No	Yes

Singleton service locator style access	No	Yes
Declarative registration of custom resource protocol handler, XML Parsers for object definitions, and type aliases	No	Yes

5.11. Configuration of IApplicationContext

Well known locations in the .NET application configuration file are used to register resource handlers, custom parsers, type alias, and custom type converts in addition to the context and objects sections mentioned previously. A sample .NET application configuration file showing all these features is shown below. Each section requires the use of a custom configuration section handler. Note that the types shown for resource handlers and parsers are fictional.

```
<configuration>
            <configSections>
                         <sectionGroup name="spring">
                                     <section name="context" type="Spring.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Support.Context.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Supp
                                     <section name="objects" type="Spring.Context.Support.D</pre>
                                    <section name="parsers" type="Spring.Context.Support.N</pre>
                                     <section name="resources" type="Spring.Context.Support</pre>
                                     <section name="typeAliases" type="Spring.Context.Suppo"</pre>
                                     <section name="typeConverters" type="Spring.Context.Su"</pre>
                               </sectionGroup>
            </configSections>
            <spring>
                         <parsers>
                                     <parser type="Spring.Aop.Config.AopNamespaceParser, Spring.AopNamespaceParser, Spring.AopNamespac
                                      <parser type="Spring.Data.Config.DatabaseNamespacePars"</pre>
                         </parsers>
                         <resources>
                                     <handler protocol="db" type="MyCompany.MyApp.Resources</pre>
                         </resources>
                         <context caseSensitive="false">
                                     <resource uri="config://spring/objects"/>
                                    <resource uri="db://user:pass@dbName/MyDefinitionsTabl">Tabl
                         </context>
```

The new sections are described below. The attribute CaseSensitive allows the for both IObjectFactory and IApplicationContext implementations to not pay attention to the case of the object names. This is important in web applications so that ASP.NET pages can be resolved in a case independent manner. The default value is true.

5.11.1. Registering custom parsers

Instead of using the default XML schema that is generic in nature to define an object's properties and dependencies, you can create your own XML schema specific to an application domain. This has the benefit of being easier to type and getting XML intellisense for the schema being used. The downside is that you need to write code that will transform this XML into Spring object definitions. One would typically implement a custom parser by deriving from the class ObjectsNamespaceParser and overriding the methods int ParseRootElement(XmlElement root, XmlResourceReader reader) and int ParseElement(XmlElement element,

XmlResourceReader reader). Registering custom parsers outside of App.config will be addressed in a future release.

To register a custom parser register a section handler of the type Spring.Context.Support.NamespaceParsersSectionHein the configSecitons section of App.config. The parser configuration section contains one or more parser> elements each with a type attribute. Below is an example that registers all the namespaces provided in Spring.



Note

As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <!-- other configuration section handler defined here
      <section name="parsers" type="Spring.Context.Support.N</pre>
     </sectionGroup>
  </configSections>
  <spring>
    <parsers>
      <parser type="Spring.Aop.Config.AopNamespaceParser, Sp</pre>
      <parser type="Spring.Data.Config.DatabaseNamespacePars"</pre>
      <parser type="Spring.Transaction.Config.TxNamespacePar</pre>
      <parser type="Spring.Validation.Config.ValidationNames"</pre>
      <parser type="Spring.Remoting.Config.RemotingNamespace"
</pre>
    </parsers>
  </spring>
</configuration>
```

You can also register custom parser programmatically using the NamespaceParserRegistry. Here is an example taken from the code used in the Transactions Quickstart application.

```
NamespaceParserRegistry.RegisterParser(typeof(DatabaseNamesp
NamespaceParserRegistry.RegisterParser(typeof(TxNamespacePar
NamespaceParserRegistry.RegisterParser(typeof(AopNamespacePar
IApplicationContext context = new XmlApplicationContext("assembly://Spring.TxQuickSt
```

5.11.2. Registering custom resource handlers

Creating a custom resource handler means implementing the IResource interface. The base class AbstractResource is a useful starting point. Look at the Spring source for classes such as FileSystemResource or AssemblyResource for implementation tips. You can register your custom resource handler either within App.config, as shown in the program listing at the start of this section using a .ResourceHandlersSectionHandler or define an object of the type

Spring.Objects.Factory.Config.ResourceHandlerCo as you would any other Spring managed object. An example of the latter is shown below:

5.11.3. Registering Type Aliases

Type aliases allow you to simplify Spring configuration file by replacing fully qualified type name with an alias for frequently used types. Aliases can be registered both within a config file and programatically and can be used anywhere in the context config file where a fully qualified type name is expected. Type aliases can also be defined for generic types.

One way to configure a type alias is to define them in a custom config section in

the Web/App.config file for your application, as well as the custom configuration section handler. See the previous XML configuration listing for an example that makes an alias for the WebServiceExporter type. Once you have aliases defined, you can simply use them anywhere where you would normally specify a fully qualified type name:

To register a type alias register a section handler of the type Spring.Context.Support.TypeAliasesSectionHandle in the configSecitons section of App.config. The type alias configuration section contains one or more <alias> elements each with a name and a type attribute. Below is an example that registers the alias for WebServiceExporter

</configuration>

For an example showing type aliases for generic types see <u>Section 5.2.5</u>, "Object <u>creation of generic types"</u>.

Another way is to define an object of the type Spring.Objects.Factory.Config.TypeAliasConfigur within the regular <objects> section of any standard Spring configuration file. This approach allows for more modularity in defining type aliases, for example if you can't access App.config/Web.config. An example of registration using a TypeAliasConfigurer is shown below

5.11.4. Registering Type Converters

TypeConverter attribute to a type definition to specify the type of the Converter. This is the preferred way of defining type converters if you control the source code for the type that you want to define a converter for. However, this configuration section allows you to specify converters for the types that you don't control, and it also allows you to override some of the standard type converters, such as the ones that are defined for some of the types in the .NET Base Class Library.

specify the App.config using You can type converters in by Spring.Context.Support.TypeConvertersSectionHandler as shown before or define the object of type Spring.Objects.Factory.Config.CustomConverterCo An example of registration using a CustomConverterConfigurer is shown below

5.12. Added functionality of the **IApplicationContext**

As already stated in the previous section, the IApplicationContext has a couple of features that distinguish it from the IObjectFactory. Let us review them one-by-one.

5.12.1. Context Hierarchies

You can structure the configuration information of application context into hierarchies that naturally reflect the internal layering of your application. As an example, abstract object definitions may appear in a parent application context configuration file, possibly as an embedded assembly resource so as not to invite accidental changes.

The nesting of <code>context</code> elements reflects the parent-child hierarchy you are creating. The nesting can be to any level though it is unlikely one would need a deep application hierarchy. The xml file must contain the <code><objects></code> as the root name. Another example of a hierarchy, but using sections in the application configuration file is shown below.

```
<configSections>
    <sectionGroup name="spring">
         <section name="context" type="Spring.Context.Support.Cor
         <section name="objects" type="Spring.Context.Support.Def
         <sectionGroup name="child">
                <section name="objects" type="Spring.Context.Support.Def
                <sectionGroup>
                </sectionGroup>
                </sectionGroup>
                 </configSections>
```

As a reminder, the type attribute of the context tag is optional and defaults to Spring.Context.Support.XmlApplicationContext. The name of the context can be used in conjunction with the service locator class, ContextRegistry, discussed in <u>Section 5.15</u>, "Service Locator access"

5.12.2. Using IMessageSource

The IApplicationContext interface extends an interface called IMessageSource and provides localization (i18n or internationalization) services for text messages and other resource data types such as images. This functionality makes it easier to use .NET's localization features at an application level and also offers some performance enhancements due to caching of retrieved resources. Together with the NestingMessageSource, capable of hierarchical message resolving, these are the basic interfaces

Spring.NET provides for localization. Let's quickly review the methods defined there:

- string GetMessage(string name): retrieves a message from the IMessageSource and using CurrentUICulture.
- string GetMessage(string name, CultureInfo cultureInfo): retrieves a message from the IMessageSource using a specified culture.
- string GetMessage(string name, params object[] args): retrieves a message from the IMessageSource using a variable list of arguments as replacement values in the message. The CurrentUICulture is used to resolve the message.
- string GetMessage(string name, CultureInfo cultureInfo, params object[] args): retrieves a message from the IMessageSource using a variable list of arguments as replacement values in the message. The specified culture is used to resolve the message.
- string GetMessage(string name, string defaultMessage, CultureInfo culture, params object[] arguments): retrieves a message from the IMessageSource using a variable list of arguments as replacement values in the message. The specified culture is used to resolve the message. If no message can be resolved, the default message is used.
- string GetMessage(IMessageSourceResolvable resolvable, CultureInfo culture) : all properties used in the methods above are also wrapped in a class the MessageSourceResolvable, which you can use in this method.
- object GetResourceObject(string name):Get a localized resource object, i.e. Icon, Image, etc. given the resource name. The CurrentUICulture is used to resolve the resource object.
- object GetResourceObject(string name, CultureInfo cultureInfo):Get a localized resource object,

i.e. Icon, Image, etc. given the resource name. The specified culture is used to resolve the resource object.

• void ApplyResources(object value, string objectName, CultureInfo cultureInfo): Uses a ComponentResourceManager to apply resources to all object properties that have a matching key name. Resource key names are of the form objectName.propertyName

When an IApplicationContext gets loaded, it automatically searches for an IMessageSource object defined in the context. The object has to have the name MessageSource. If such an object is found, all calls to the methods described above will be delegated to the message source that was found. If no message source was found, the IApplicationContext checks to see if it has a parent containing a similar object, with a similar name. If so, it uses that object as the IMessageSource. If it can't find any source for messages, an empty StaticMessageSource will be instantiated in order to be able to accept calls to the methods defined above.

Fallback behavior

The fallback rules for localized resources seem to have a bug that is fixed by applying Service Pack 1 for .NET 1.1. This affects the use of IMessageSource.GetMessage methods that specify CultureInfo. The core of the issue in the .NET BCL is the method ResourceManager.GetObject that accepts CultureInfo.

Spring.NET provides two IMessageSource implementations. These are ResourceSetMessageSource and StaticMessageSource. IHierarchicalMessageSource implement Both messages hierarchically. The StaticMessageSource is hardly ever used but provides programmatic ways to add messages to the source. The ResourceSetMessageSource is more interesting and an example is provided for in the distribution and discussed more extensively in the Chapter 33. ResourceSetMessageSource The section. IoC Quickstarts configured by providing a list of ResourceManagers. When a message code is to be resolved, the list of ResourceManagers is searched to resolve the code. For each ResourceManager a ResourceSet is retrieved and asked to resolve the code. Note that this search does not replace the standard hub-and-spoke search for localized resources. The ResourceManagers list specifies the multiple 'hubs' where the standard search starts.

You can specify the arguments to construct a ResourceManager as a two part string value containing the base name of the resource and the assembly name. This will be converted to a ResourceManager via the ResourceManagerConverter TypeConverter. This converter can be similarly used to set a property on any object that is of the type ResourceManager. You may also specify an instance of the ResourceManager to use via an object reference. The convenience class Spring.Objects.Factory.Config.ResourceManagerFacan be used to conveniently create an instance of a ResourceManager.

In application code, a call to <code>GetMessage</code> will retrieve a properly localized message string based on a code value. Any arguments present in the retrieved string are replaced using <code>String.Format</code> semantics. The ResourceManagers, ResourceSets and retrieved strings are cached to provide quicker lookup performance. The key 'HelloMessage' is contained in the resource file with a value of <code>Hello {0} {1}</code>. The following call on the application context will return the string <code>Hello Mr. Anderson</code>. Note

that the caching of ResourceSets is via the concatenation of the ResourceManager base name and the CultureInfo string. This combination must be unique.

It is possible to chain the resolution of messages by passing arguments that are themselves messages to be resolved giving you greater flexibility in how you can structure your message resolution. This is achieved by passing as an argument a class that implements IMessageResolvable instead of a string literal. The convenience class DefaultMessageResolvable is available for this purpose. As an example if the resource file contains a key name error.required that has the value '{0} is required {1}' and another key name field.firstname with the value 'First name'. The following code will create the string 'First name is required dude!'

The examples directory in the distribution contains an example program, Spring.Examples.AppContext, that demonstrates usage of these features.

The IMessageSourceAware interface can also be used to acquire a reference to any IMessageSource that has been defined. Any object that is defined in an IApplicationContext that implements the IMessageSourceAware interface will be injected with the application context's IMessageSource when it (the object) is being created and configured.

5.12.3. Using resources within Spring.NET

A lot of applications need to access resources. Resources here, might mean files, but also news feeds from the Internet or normal web pages. Spring.NET provides a clean and transparent way of accessing resources in a protocol independent way. The IApplicationContext has a method (GetResource(string)) to take care of this. Refer to Section 7.1, "Introduction" for more information on the string format to use and the IResource abstraction in general.

5.12.4. Loosely coupled events

The Eventing Registry allows developers to utilize a loosely coupled event wiring mechanism. By decoupling the event publication and the event subscription, most of the mundane event wiring is handled by the IoC container. Event publishers can publish their event to a central registry, either all of their events or a subset based on criteria such as delegate type, name, return value, etc... Event subscribers can choose to subscribe to any number of published events. Subscribers can subscriber to events based on the type of object exposing them, allowing one subscriber to handle all events of a certain type without regards to how many different instances of that type are created.

The Spring.Objects.Events.IEventRegistry interface represents the central registry and defines publish and subscribe methods.

- void PublishEvents(object sourceObject): publishes all events of the source object to subscribers that implement the correct handler methods.
- void Subscribe(object subscriber): The subscriber receives all events from the source object for which it has matching handler methods.
- void Subscribe(object subscriber, Type targetSourceType): The subscriber receives all events from a source object of a particular type for which it has matching handler methods.
- void Unsubscribe(object subscriber): Unsubscribe all events from the source object for which it has matching handler methods.
- void Unsubscribe(object subscriber, Type

targetSourceType): Unsubscribe all events from a source object of a particular type for which it has matching handler methods.

IApplicationContext implements this interface and delegates the implementation to an instance of Spring.Objects.Events.Support.EventRegistry. You are free to create and use as many EventRegistries as you like but since it is common to use only one in an application, IApplicationContext provides convenient access to a single instance.

Within

example/Spring/Spring.Examples.EventRegistry directory you will find an example on how to use this functionality. When you open up the project, the most interesting file is the EventRegistryApp.cs file. This application loads a set of object definitions from the application configuration file into an IApplicationContext instance. From there, three objects are loaded up: one publisher and two subscribers. The publisher publishes its events to the IApplicationContext instance:

```
// Create the Application context using configuration file
IApplicationContext ctx = ContextRegistry.GetContext();

// Gets the publisher from the application context
MyEventPublisher publisher = (MyEventPublisher)ctx.GetObject

// Publishes events to the context.
ctx.PublishEvents( publisher );
```

One of the two subscribers subscribes to all events published to the IApplicationContext instance, using the publisher type as the filter criteria.

```
// Gets first instance of subscriber
MyEventSubscriber subscriber = (MyEventSubscriber)ctx.GetObj
// Gets second instance of subscriber
MyEventSubscriber subscriber2 = (MyEventSubscriber)ctx.GetOb
// Subscribes the first instance to the any events published ctx.Subscribe( subscriber, typeof(MyEventPublisher) );
```

This will wire the first subscriber to the original event publisher. Anytime the event publisher fires an event, (publisher.ClientMethodThatTriggersEvent1();) the first subscriber will handle the event, but the second subscriber will not. This allows for selective subscription, regardless of the original prototype definition.

5.12.5. Event notification from **IApplicationContext**

Event handling in the IApplicationContext is provided through the IApplicationListener interface that contains the single method void OnApplicationEvent(object source, ApplicationEventArgs applicationEventArgs). Classes that implement the IApplicationListener interface are automatically registered as a listener with the IApplicationContext. Publishing an event is done via the context's PublishEvent(ApplicationEventArgs eventArgs) method. This implementation is based on the traditional Observer design pattern.

The event argument type, ApplicationEventArgs, adds the time of the event firing as a property. The derived class ContextEventArgs is used to notify observers on the lifecycle events of the application context. It contains a property ContextEvent Event that returns the enumeration Refreshed or Closed. The Refreshed enumeration value indicated that the IApplicationContext was either initialized or refreshed. Initialized here means that all objects are loaded, singletons are pre-instantiated and the IApplicationContext is ready for use. The Closed is published when the IApplicationContext is closed using the Dispose() method on the IConfigurableApplicationContext interface. Closed here means that singletons are destroyed.

Implementing custom events can be done as well. Simply call the PublishEvent method on the IApplicationContext, specifying a parameter which is an instance of your custom event argument subclass.

Let's have a look at an example. First, the IApplicationContext:

and then, the actual objects:

```
public class EmailObject : IApplicationContextAware {
    // the blackList
    private IList blackList;

public IList BlackList
    {
        set { this.blackList = value; }
    }

public IApplicationContext ApplicationContext
    {
        set { this.ctx = value; }
    }

public void SendEmail(string address, string text) {
        if (blackList.contains(address))
        {
            BlackListEvent evt = new BlackListEvent(address, ctx.publishEvent(evt);
            return;
        }
}
```

```
// send email...
}

public class BlackListNotifier : IApplicationListener
{

    // notification address
    private string notificationAddress;

public string NotificationAddress
    {
        set { this.notificationAddress = value; }
}

public void OnApplicationEvent(ApplicationEvent evt)
    {
        if (evt instanceof BlackListEvent)
        {
            // notify appropriate person
        }
     }
}
```

5.13. Customized behavior in the ApplicationContext

The IObjectFactory already offers a number of mechanisms to control the lifecycle of objects deployed in it (such as marker interfaces like IInitializingObject and System.IDisposable, their configuration only equivalents such as init-method and destroy-method) attributes in an XmlObjectFactory configuration, and object post-processors. In an IApplicationContext, all of these still work, but additional mechanisms are added for customizing behavior of objects and the container.

5.13.1. The **IApplicationContextAware** marker interface

All marker interfaces available with ObjectFactories still work. The IApplicationContext does add one extra marker interface which objects may implement, IApplicationContextAware. An object which implements this interface and is deployed into the context will be called back on creation of the object, using the interface's ApplicationContext property, and provided with a reference to the context, which may be stored for later interaction with the context.

5.13.2. The IObjectPostProcessor

classes which Object post-processors implement are Spring.Objects.Factory.Config.IObjectPostProces interface, have already been mentioned. It is worth mentioning again here though, convenient that post-processors are much more to use in IApplicationContexts **IObjectFactory** in plain than instances. In an IApplicationContext, any deployed object which implements the above marker interface is automatically detected and registered as an object post-processor, to be called appropriately at creation time for each object in the factory.

5.13.3. The IObjectFactoryPostProcessor

Object factory post-processors are classes which implement the

Spring.Objects.Factory.Config.IObjectFactoryPos interface, have <u>already</u> been mentioned. It is worth mentioning again here though, that object factory post-processors are much more convenient to use in IApplicationContexts. In an IApplicationContext, any deployed object which implements the above marker interface is automatically detected as an object factory post-processor, to be called at the appropriate time.

5.13.4. The PropertyPlaceholderConfigurer

The PropertyPlaceholderConfigurer has already been described in the context of its use within an IObjectFactory. It is worth mentioning here though, that it is generally more convenient to use it with an IApplicationContext, since the context will automatically recognize and apply any object factory post-processors, such as this one, when they are simply deployed into it like any other object. There is no need for a manual step to execute it.

5.14. Configuration of ApplicationContext without using XML

The class GenericApplicationContext can be used as a basis for creating an IApplicationContext implementation that read the container metadata from sources other than XML. This could be by scanning objects in a .DLL for known attributes or a scripting language that leverages a DSL to create terse IObjectDefinitions. There is a class, Spring.Objects.Factory.Support.ObjectDefinitionBuilder offers some convenience methods for creating an IObjectDefinition in a less verbose manner than using the RootObjectDefinition API. The following shows how to configure the GenericApplicationContext to read from XML, just so show familiar API usage

```
GenericApplicationContext ctx = new GenericApplicationContext XmlObjectDefinitionReader reader = new XmlObjectDefinitionReader.LoadObjectDefinitions("assembly://Spring.Core.Tests/Sreader.LoadObjectDefinitions("assembly://Spring.Core.Tests/Sreader.LoadObjectDefinitions("assembly://Spring.Core.Tests/Sreader.LoadObjectDefinitions("assembly://Spring.Core.Tests/Sctx.Refresh();
```

The implementation of IObjectDefinitionReader is responsible for creating the configuration metadata, i.e., implementations of RootObjectDefinition, etc. Note a web version of this application class has not yet been implemented.

An example, with a *yet to be created* DLL scanner, that would get configuration metadata from the .dll named MyAssembly.dll located in the runtime path, would look something like this

```
GenericApplicationContext ctx = new GenericApplicationContex
ObjectDefinitionScanner scanner = new ObjectDefinitionScanner
scanner.scan("MyAssembly.dll");
ctx.refresh();
```

Refer to the Spring API documentation for more information.

5.15. Service Locator access

The majority of the code inside an application is best written in a Dependency Injection (Inversion of Control) style, where that code is served out of an IObjectFactory or IApplicationContext container, has its own dependencies supplied by the container when it is created, and is completely unaware of the container. However, there is sometimes a need for singleton (or quasi-singleton) style access to an IObjectFactory or **IApplicationContext**. For example, third party code may try to construct a new object directly without the ability to force it to get these objects out of the IObjectFactory. Similarly, nested user control components in a WinForms application are created inside the generated code InitializeComponent. If this user control would like to obtain references to objects contained in the container it can use the service locator style approach and 'reach out' from inside the code to obtain the object it requires. (Note support for DI in WinForms is under development.)

Spring.Context.Support.ContextRegistry The allows you to obtain a reference to an IApplicationContext via a static locator method. The ContextRegistry is initialized when creating an IApplicationContext through use of the ContextHandler discussed previously. The simple static method GetContext() can then be used to retrieve the context. Alternatively, if you create IApplicationContext though other means you can register it with the ContextRegistry via the method void RegisterContext(IApplicationContext context) in the start-up code of your application. Hierarchical context retrieval is also supported though the use of the GetContext(string name) method, for example:

IApplicationContex ctx = ContextRegistry.GetContext("mySubContext")

This would retrieve the nested context for the context configuration shown previously.

<spring>
 <context>

Do not call ContextRegistry.GetContext within a constructor as it will result in and endless recursion. (This is scheduled to be fixed in 1.1.1) In this case it is quite likely you can use the IApplicationContextAware interface and then retrieve other objects in a service locator style inside an initialization method.

The ContextRegistry.Clear() method will remove all contexts. On .NET 2.0, this will also call the ConfigurationManager's RefreshSection method so that the Spring context configuration section will be reread from disk when it is retrieved again. Note that in a web application RefeshSection will not work as advertised and you will need to touch the web.config files to reload a configuration.

5.16. Stereotype attributes

Beginning with Spring 1.2, the [Repository] attribute was introduced as a marker for any class that fulfills the role or stereotype of a repository (a.k.a. Data Access Object or DAO). Among the possibilities for leveraging such a marker is the automatic translation of exceptions as described in Exception Translation.

Spring 1.2 introduces further stereotype annotations: [Component] and [Service]. [Component] serves as a generic stereotype for any Spring-managed component; whereas, [Repository] and [Service] serve as specializations of [Component] for more specific use cases (e.g., in the persistence and service layers, respectively). The ASP.NET MVC [Controller] attribute will serve this purpose for the controller layer. What this means is that you can annotate your component classes with [Component], but by annotating them with [Repository] or [Service] your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. Of course, it is also possible that [Repository] and [Service] may carry additional semantics in future releases of the Spring Framework. Thus, if you are making a decision between using [Component] or [Service] for your service layer, [Service] is clearly the better choice. Similarly, as stated above, [Repository] is already supported as a marker for automatic exception translation in your persistence layer. The next version of Spring will use the [Component] attribute to perform attribute based autowiring by-type as in the Spring Java Framework.

^[1] See the section entitled <u>Section 3.1, "Inversion of Control"</u>

^[2] More information about assembly names can be found in the *Assembly Names* section of the .NET Framework Developer's Guide (installed as part of the .NET SDK), or online at Microsoft's MSDN website, by searching for *Assembly Names*.

^[3] More information about creating custom **TypeConverter** implementations can be found online at Microsoft's MSDN website, by searching for *Implementing a Type Converter*.

Chapter 6. The IObjectWrapper and Type conversion

6.1. Introduction

The concepts encapsulated by the <code>IObjectWrapper</code> interface are fundamental to the workings of the core Spring.NET libraries The typical application developer most probably will not ever have the need to use the <code>IObjectWrapper</code> directly... because this is reference documentation however, we felt that some explanation of this core interface might be right. The <code>IObjectWrapper</code> is explained in this chapter since if you were going to use it at all, you would probably do that when trying to bind data to objects, which, nicely enough, is precisely the area that the <code>IObjectWrapper</code> addresses.

6.2. Manipulating objects using the IObjectWrapper

One quite important concept of the Spring. Objects namespace is encapsulated in the definition IObjectWrapper interface and its ObjectWrapper implementation, the corresponding class. The functionality offered by the IObjectWrapper includes methods to set and get property values (either individually or in bulk), get property descriptors (instances of the System.Reflection.PropertyInfo class), and readability and writability query the of properties. The **IObjectWrapper** also offers support for nested properties, enabling the setting of properties on subproperties to an unlimited depth. **IObjectWrapper** usually isn't used by application code directly, but by framework classes such as the various IObjectFactory implementations.

The way the **IObjectWrapper** works is partly indicated by its name: *it* wraps an object to perform actions on a wrapped object instance... such actions would include the setting and getting of properties exposed on the wrapped object.

Note: the concepts explained in this section are not important to you if you're not planning to work with the **IObjectWrapper** directly.

6.2.1. Setting and getting basic and nested properties

Setting and getting properties is done using the SetPropertyValue() and GetPropertyValue() methods, for which there are a couple of overloaded variants. The details of the various overloads (including return values and method parameters) are all described in the extensive API documentation supplied as a part of the Spring.NET distribution.

The aforementioned SetPropertyValue() and GetPropertyValue() methods have a number of conventions for indicating the path of a property. A property path is an expression that implementations of the IObjectWrapper interface can use to look up the properties of the wrapped object; some examples of property paths include...

Table 6.1. Examples of property paths

Path	Explanation	
name	Indicates the Name property of the wrapped object.	
account.name	Indicates the nested property name of the account property of the wrapped object.	
account[2]	Indicates the <i>third</i> element of the account property of the wrapped object. Indexed properties are typically collections such as lists and dictionaries, but can be any class that exposes an indexer.	

Below you'll find some examples of working with the **IObjectWrapper** to get and set properties. Consider the following two classes:

```
[C#]
public class Company
{
    private string name;
    private Employee managingDirector;

public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }

public Employee ManagingDirector
    {
        get { return this.managingDirector; }
        set { this.managingDirector = value; }
}
```

```
[C#]
public class Employee
{
    private string name;
    private float salary;

public string Name
{
```

```
get { return this.name; }
    set { this.name = value; }
}

public float Salary
{
    get { return salary; }
    set { this.salary = value; }
}
```

The following code snippets show some examples of how to retrieve and manipulate some of the properties of IObjectWrapper-wrapped Company and Employee instances.

```
[C#]
Company c = new Company();
IObjectWrapper owComp = new ObjectWrapper(c);
// setting the company name...
owComp.SetPropertyValue("name", "Salina Inc.");
// can also be done like this...
PropertyValue v = new PropertyValue("name", "Salina Inc.");
owComp.SetPropertyValue(v);

// ok, let's create the director and bind it to the company.
Employee don = new Employee();
IObjectWrapper owDon = new ObjectWrapper(don);
owDon.SetPropertyValue("name", "Don Fabrizio");
owComp.SetPropertyValue("managingDirector", don);

// retrieving the salary of the ManagingDirector through the float salary = (float)owComp.GetPropertyValue("managingDirector")
```

Note that since the various Spring.NET libraries are compliant with the Common Language Specification (CLS), the resolution of arbitrary strings to properties, events, classes and such is performed in a case-insensitive fashion. The previous examples were all written in the C# language, which is a case-sensitive language, and yet the Name property of the Employee class was set using the all-lowercase 'name' string identifier. The following example (using the classes defined previously) should serve to illustrate this...

```
[C#]
// ok, let's create the director and bind it to the company.
Employee don = new Employee();
IObjectWrapper owDon = new ObjectWrapper(don);
owDon.SetPropertyValue("naMe", "Don Fabrizio");
owDon.GetPropertyValue("nAmE"); // gets "Don Fabrizio"

IObjectWrapper owComp = new ObjectWrapper(new Company());
owComp.SetPropertyValue("ManaGINGdirecToR", don);
owComp.SetPropertyValue("mANaGiNgdirector.salARY", 80000);
Console.WriteLine(don.Salary); // puts 80000
```

The case-insensitivity of the various Spring.NET libraries (dictated by the CLS) is not usually an issue... if you happen to have a class that has a number of properties, events, or methods that differ only by their case, then you might want to consider refactoring your code, since this is generally regarded as poor programming practice.

6.2.2. Other features worth mentioning

In addition to the features described in the preceding sections there a number of features that might be interesting to you, though not worth an entire section.

- determining readability and writability: using the IsReadable() and IsWritable() methods, you can determine whether or not a property is readable or writable.
- retrieving PropertyInfo instances: using GetPropertyInfo(string) and GetPropertyInfos() you can retrieve instances of the System.Reflection.PropertyInfo class, that might come in handy sometimes when you need access to the property metadata specific to the object being wrapped.

6.3. Type conversion

If you associate a TypeConverter with the definition of a custom Type using the standard .NET mechanism (see the example code below), Spring.NET will use the associated TypeConverter to do the conversion.

```
[C#]
[TypeConverter (typeof (FooTypeConverter))]
public class Foo
{
}
```

The TypeConverter class from the System.ComponentModel namespace of the .NET BCL is used extensively by the various classes in the Spring.Core library, as said class "... provides a unified way of converting types of values to other types, as well as for accessing standard values and subproperties." [4]

For example, a date can be represented in a human readable format (such as **30th** August 1984), while we're still able to convert the human readable form to the original date format or (even better) to an instance of the System. DateTime class. This behavior can be achieved by using the standard of .NET idiom decorating class with a TypeConverterAttribute. Spring.NET also offers another means of associating a TypeConverters with a class. You might want to do this to achieve a conversion that is not possible using standard idiom... for example, the Spring.Core library contains a custom TypeConverter converts comma-delimited strings to String array instances. Registering custom converters on an IObjectWrapper instance gives the wrapper the knowledge of how to convert properties to the desired Type.

An example of where property conversion is used in Spring.NET is the setting of properties on objects, accomplished using the aforementioned TypeConverters. When mentioning System.String as the value of a property of some object (declared in an XML file for instance), Spring.NET will (if the type of the associated property is System.Type) use the RuntimeTypeConverter class to try to resolve the property value to a

Type object. The example below demonstrates this automatic conversion of the Example.Xml.SAXParser (a string) into the corresponding Type instance for use in this factory-style class.

```
<objects xmlns="http://www.springframework.net">
<object id="parserFactory" type="Example.XmlParserFactory, E
destroy-method="Close">
    <property name="ParserClass" value="Example.Xml.SAXParser,
    </object>
    </objects>
```

```
[C#]
public class XmlParserFactory
{
    private Type parserClass;

    public Type ParserClass
    {
        get { return this.parserClass; }
        set { this.parserClass = value; }
    }

    public XmlParser GetParser ()
    {
        return Activator.CreateInstance (ParserClass);
    }
}
```

6.3.1. Type Conversion for Enumerations

The default type converter for enumerations is the System.ComponentModel.EnumConverter class. To specify the value for an enumerated property, simply use the name of the property. For example the TestObject class has a property of the enumerated type FileMode. One of the values for this enumeration is named Create. The following XML fragment shows how to configure this property

```
<object id="rod" type="Spring.Objects.TestObject, Spring.Cor
  <pre>property name="name" value="Rod"/>
```

```
///
```

6.4. Built-in TypeConverters

Spring.NET has a number of built-in TypeConverters to make life easy. Each of those is listed below and they are all located in the Spring.Objects.TypeConverters namespace of the Spring.Core library.

Table 6.2. Built-in TypeConverters

Туре	Explanation
RuntimeTypeConverter	Parses strings representing System. Types to a System. Types and the other way around.
FileInfoConverter	Capable of resolving strings to a System.IO.FileInfo object.
StringArrayConverter	Capable of resolving a comma-delimited list of strings string-array and vice versa.
UriConverter	Capable of resolving a string representation of a URI to actual Uri-object.
FileInfoConverter	Capable of resolving a string representation of a FileIn an actual FileInfo-object.
StreamConverter	Capable of resolving Spring IResource URI (string) to corresponding InputStream-object.
ResourceConverter	Capable of resolving Spring IResource URI (string) to IResource object.
ResourceManagerConverter	Capable of resolving a two part string (resource name, assembly name) to a System.Resources.ResourceMana object.
RgbColorConverter	Capable of resolving a comma separated list of Red, G Blue integer values to a System.Drawing.Color structure.
RegexConverter	Converts string representation of regular expression in instance of System.Text.RegularExpressions.Regex

Spring.NET uses the standard .NET mechanisms for the resolution of

System.Types, including, but not limited to checking any configuration files associated with your application, checking the Global Assembly Cache (GAC), and assembly probing.

6.4.1. Custom type converters

You can register a custom type converter either Programatically using the class TypeConverterRegistry or through configuration of Spring's container and described in the section Registering Type Converters.

[4] More information about creating custom **TypeConverter** implementations can be found online at Microsoft's MSDN website, by searching for *Implementing a Type Converter*.

Chapter 7. Resources

7.1. Introduction

The IResource interface contained in the Spring.Core.IO namespace provides a common interface to describe and access data from diverse resource locations. This abstraction lets you treat the InputStream from a file and from a URL in a polymorphic and protocol-independent manner... the .NET BCL does not provide such an abstraction. The IResource interface inherits from IInputStream that provides a single property Stream InputStream. The IResource interface adds descriptive information about the resource via a number of additional properties. Several implementations for common resource locations, i.e. file, assembly, uri, are provided and you may also register custom IResource implementations.

7.2. The IResource interface

The IResource interface is shown below

```
public interface IResource : IInputStreamSource
{
  bool IsOpen { get; }

  Uri Uri { get; }

  FileInfo File { get; }

  string Description { get; }

  bool Exists { get; }

  IResource CreateRelative(string relativePath);
}
```

Table 7.1. IResource Properties

Property	Explanation
InputStream	Inherited from IInputStream. Opens and returns a System. IO.Stream. It is expected that each invocation returns a fresh Stream. It is the responsibility of the caller to close the stream.
Exists	returns a boolean indicating whether this resource actually exists in physical form.
Is0pen	returns a boolean indicating whether this resource represents a handle with an open stream. If true, the InputStream cannot be read multiple times, and must be read once only and then closed to avoid resource leaks. Will be false for all usual resource implementations, with the exception of InputStreamResource.
Description	Returns a description of the resource, such as the fully qualified file name or the actual URL.
Uri	The Uri representation of the resource.
File	Returns a System.IO.FileInfO for this resource if it can be resolved to an absolute file path.

and the methods

Table 7.2. IResource Methods

Method	Explanation
IResource CreateRelative (string relativePath)	Creates a resource relative to this resource using relative path like notation (./ and/).

You can obtain an actual URL or File object representing the resource if the underlying implementation is compatible and supports that functionality.

The Resource abstraction is used extensively in Spring itself, as an argument type in many method signatures when a resource is needed. Other methods in some Spring APIs (such as the constructors to various IApplicationContext implementations), take a String which is used to create a Resource appropriate to that context implementation

While the Resource interface is used a lot with Spring and by Spring, it's actually very useful to use as a general utility class by itself in your own code, for access to resources, even when your code doesn't know or care about any other parts of Spring. While this couples your code to Spring, it really only couples it to this small set of utility classes and can be considered equivalent to any other library you would use for this purpose

7.3. Built-in IResource implementations

The resource implementations provided are

- AssemblyResource accesses data stored as .NET resources inside an assembly. Uri syntax is assembly://<AssemblyName>/<NameSpace>/<Resources
- ConfigSectionResource accesses Spring.NET configuration data stored in a custom configuration section in the .NET application configuration file (i.e. App.config). Uri syntax is config://<path to section>
- FileSystemResource accesses file system data. Uri syntax is file://<filename>
- InputStreamResource a wrapper around a raw System.IO.Stream. Uri syntax is not supported.
- UriResource accesses data from the standard System.Uri protocols such as http and https. In .NET 2.0 you can use this also for the ftp protocol. Standard Uri syntax is supported.

Refer to the MSDN documentation for more information on <u>supported Uri scheme</u> <u>types</u>.

7.3.1. Registering custom IResource implementations

The configuration section handler, ResourceHandlersSectionHandler, is used to register any custom IResource implementations you have created. In the configuration section you list the type of IResource implementation and the protocol prefix. Your custom IResource implementation must provide a constructor that takes a string as it's sole argument that represents the URI string. Refer to the SDK documentation for ResourceHandlersSectionHandler information. of more An example the ResourceHandlersSectionHandler is shown below fictional IResource implementation that interfaces with a database.

7.4. The IResourceLoader

To load resources given their Uri syntax, an implementation of the IResourceLoader is used. The default implementation is ConfigurableResourceLoader. Typically you will not need to access this class directly since the IApplicationContext implements the IResourceLoader interface that contains the single method IResource GetResource(string location). The provided implementations of IApplicationContext delegate this method to an instance of ConfigurableResourceLoader which supports the Uri protocols/schemes listed previously. If you do not specify a protocol then the file protocol is used. The following shows some sample usage.

```
IResource resource = appContext.GetResource("http://www.spri
resource = appContext.GetResource("assembly://Spring.Core.Te
resource = appContext.GetResource("https://sourceforge.net/"
resource = appContext.GetResource("file:///C:/WINDOWS/ODBC.I

StreamReader reader = new StreamReader(resource.InputStream)
Console.WriteLine(reader.ReadToEnd());
```

Other protocols can be registered along with a new implementations of an IResource that must correctly parse a Uri string in its constructor. An example of this can be seen in the Spring.Web namespace that uses Server.MapPath to resolve the filename of a resource.

The **CreateRelative** method allows you to easily load resources based on a relative path name. In the case of relative assembly resources, the relative path navigates the namespace within an assembly. For example:

```
IResource res = new AssemblyResource("assembly://Spring.Core
IResource res2 = res.CreateRelative("./IO/TestIOResource.txt
```

This loads the resource TestResource.txt and then navigates to the Spring.Core.IO namespace and loads the resource TestIOResource.txt

7.5. The IResourceLoaderAware interface

The IResourceLoaderAware interface is a special marker interface, identifying objects that expect to be provided with a IResourceLoader reference.

```
public interface IResourceLoaderAware
{
   IResourceLoader ResourceLoader
   {
      set;
      get;
   }
}
```

When a class implements IResourceLoaderAware and is deployed into an application context (as a Spring-managed object), it is recognized as IResourceLoaderAware by the application context. The application context will then invoke the ResourceLoader property, supplying itself as the argument (remember, all application contexts in Spring implement the IResourceLoader interface).

IApplicationContext Of since is course, an a IResourceLoader, object implement the could also the IApplicationContextAware interface and use the application context directly to load resources, but in general, it's better to use the specialized IResourceLoader interface if that's all that's needed. The code would just be coupled to the resource loading interface, which can be considered utility interface, whole a and not the Spring IApplicationContext interface.

7.6. Application contexts and IResource paths

An application context constructor (for a specific application context type) generally takes a string or array of strings as the location path(s) of the resource(s) such as XML files that make up the definition of the context. For example, you can create an XmlApplicationContext from two resources as follows:

```
IApplicationContext context = new XmlApplicationContext(
    "file://objects.xml", "assembly://MyAssembly/MyProje
```

Chapter 8. Threading and Concurrency Support

8.1. Introduction

The purpose of the Spring. Threading namespace is to provide a place to keep useful concurrency abstractions that augment those in the BCL. Since Doug Lea has provided a wealth of mature public domain concurrency abstractions in his Java based 'EDU.oswego.cs.dl.util.concurrent' libraries we decided to port a few of his abstractions to .NET. So far, we've only ported three classes, the minimum necessary to provide basic object pooling functionality to support an AOP based pooling aspect and to provide a Semaphore class that was mistakenly not included in .NET 1.0/1.1.

There is also an important abstraction, IThreadStorage, for performing thread local storage.

8.2. Thread Local Storage

Depending on your runtime environment there are different strategies to use for storing objects in thread local storage. If you are in web applications a single Request may be executed on different threads. As such, the location to store thread local objects is in <code>HttpContext.Current</code>. For other environments

System.Runtime.Remoting.Messaging.CallContext is used. For more background information on the motivation behind these choices, say as compared to the attribute [ThreadStatic] refer to "Piers7"'s blog and this forum post. The interface IThreadStorage serves as the basis for the thread local storage abstraction and various implementations can be selected from depending on your runtime requirements. Configuring the implementation of IThreadStorage makes it easier to have more portability across runtime environments.

The API is quite simple and shown below

```
public interface IThreadStorage
{
  object GetData(string name)

  void SetData(string name, object value)

  void FreeNamedDataSlot(string name)
}
```

The methods **GetData** and **SetData** are responsible for retrieving and setting the object that is to be bound to thread local storage and associating it with a name. Clearing the thread local storage is done via the method **FreeNamedDataSlot**.

In Spring. Core is the implementation, CallContextStorage, that CallContext also implementation directly uses and the LogicalThreadContext which default by uses CallContextStorage but can be configured via the static method SetStorage(IThreadStorage). The methods on

CallContextStorage and LogicalThreadContext are static.

In Spring.Web is the implementation HttpContextStorage which HttpContext to thread local data the store and HybridContextStorage that uses HttpContext if within a web HttpContext.Current != environment, i.e. null, CallContext otherwise.

Spring internally uses LogicalThreadContext as this doesn't require a coupling to the System. Web namespace. In the case of Spring based web applications, Spring's WebSupportModule sets the storage strategy of LogicalThreadContext to be HybridContextStorage.

8.3. Synchronization Primitives

When you take a look at these synchronization classes, you'll wonder why it's even necessary when <code>System.Threading</code> provides plenty of synchronization options. Although <code>System.Threading</code> provides great synchronization classes, it doesn't provide well-factored abstractions and interfaces for us. Without these abstractions, we will tend to code at a low-level. With enough experience, you'll eventually come up with some abstractions that work well. Doug Lea has already done a lot of that research and has a class library that we can take advantage of.

8.3.1. ISync

ISync is the central interface for all classes that control access to resources from multiple threads. It's a simple interface which has two basic use cases. The first case is to block indefinitely until a condition is met:

```
void ConcurrentRun(ISync lock) {
  lock.Acquire(); // block until condition met
  try {
     // ... access shared resources
  }
  finally {
    lock.Release();
  }
}
```

The other case is to specify a maximum amount of time to block before the condition is met:

```
void ImpatientConcurrentRun(ISync lock) {
   // block for at most 10 milliseconds for condition
   if ( lock.Attempt(10) ) {
      try {
            // ... access shared resources
         }
      finally {
            lock.Release();
          }
      } else {
```

```
// complain of time out
}
```

8.3.2. SyncHolder

The SyncHolder class implements the System.IDisposable interface and so provides a way to use an ISync with the using C# keyword: the ISync will be automatically Acquired and then Released on exiting from the block.

This should simplify the programming model for code using (!) an **ISync**:

```
ISync sync = ...
using (new SyncHolder(sync))
{
    // ... code to be executed
    // holding the ISync lock
}
```

There is also the timed version, a little more cumbersome as you must deal with timeouts:

```
ISync sync = ...
long msecs = 100;
...
// try to acquire the ISync for msecs milliseconds
try
{
   using (new SyncHolder(sync, msecs))
   {
      // ... code to be executed
      // holding the ISync lock
   }
}
catch (TimeoutException)
{
   // deal with failed lock acquisition
}
```

8.3.3. Latch

The Latch class implements the ISync interface and provides an implementation of a *latch*. A latch is a boolean condition that is set at most once, ever. Once a single release is issued, all acquires will pass. It is similar to a ManualResetEvent initialized unsignalled (Reset) and can only be Set(). A typical use is to act as a start signal for a group of worker threads.

```
class Boss {
  Latch _startPermit;
  void Worker() {
    // very slow worker initialization ...
   // ... attach to messaging system
    // ... connect to database
    startPermit.Acquire();
    // ... use resources initialized in Mush
    // ... do real work
 }
 void Mush() {
    _startPermit = new Latch();
    for (int i=0; i<10; ++i) {
      new Thread(new ThreadStart(Worker)).Start();
    // very slow main initialization ...
    // ... parse configuration
    // ... initialize other resources used by workers
    _startPermit.Release();
 }
}
```

8.3.4. Semaphore

The **Semaphore** class implements the **ISync** interface and provides an implementation of a semaphore. Conceptually, a semaphore maintains a set of

permits. Each Acquire() blocks if necessary until a permit is available, and then takes it. Each Release() adds a permit. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly. A typical use is to control access to a pool of shared objects.

```
class LimitedConcurrentUploader {
    // ensure we don't exceed maxUpload simultaneous uploads
    Semaphore _available;
    public LimitedConcurrentUploader(maxUploads) {
        _available = new Semaphore(maxUploads);
    }
    // no matter how many threads call this method no more
    // than maxUploads concurrent uploads will occur.
    public Upload(IDataTransfer upload) {
        _available.Acquire();
        try {
            upload.TransferData();
        }
        finally {
            _available.Release();
      }
    }
}
```

Chapter 9. Object Pooling

9.1. Introduction

The Spring.Pool namespace contains a generic API for implementing pools of objects. Object pooling is a well known technique to minimize the creation of objects that can take a significant amount of time. Common examples are to create a pool of database connections such that each request to the database can reuse an existing connection instead of creating one per client request. Threads are also another common candidate for pooling in order to increase responsiveness of an application to multiple concurrent client requests.

.NET contains support for object pooling in these common scenarios. Support for database connection pools is directly supported by ADO.NET data providers as a configuration option. Similarly, thread pooling is supported via the System.ThreadPool class. Support for pooling of other objects can be done using the CLR managed API to COM+ found in the System.EnterpriseServices namespace.

Despite this built-in support there are scenarios where you would like to use alternative pool implementations. This may be because the default implementations, such as System.ThreadPool, do not meet your requirements. (For a discussion on advanced ThreadPool usage see Smart Thread Pool by Ami Bar.) Alternatively, you may want to pool classes that do not inherit from System.EnterpriseServices.ServicedComponent.

Instead of making changes to the object model to meet this inheritance requirement, Spring .NET provides similar support for pooling, but for any object, by using AOP proxies and a generic pool API for managing object instances.

Note, that if you are concerned only with applying pooling to an existing object, the pooling APIs discussed here are not very important. Instead the use and configuration

Spring.Aop.Target.SimplePoolTargetSource is more relevant. Pooling of objects can either be done Programatically or through the XML configuration of the Spring .NET container. Attribute support for pooling, similar to the ServicedComponent approach, will be available in a future release of Spring.NET.

<u>Chapter 33. *IoC Quickstarts*</u> contains an example that shows the use of the pooling API independent of AOP functionality.

9.2. Interfaces and Implementations

The Spring.Pool namespace provides two simple interfaces to manage pools of objects. The first interface, IObjectPool describes how to take and put back an object from the pool. The second interface IPoolableObjectFactory is meant to be used in conjunction with implementations of the IObjectPool to provide guidance in calling various lifecycle events on the objects managed by the pool. These interfaces are based Jakarta Commons the Pool API. on Spring.Pool.Support.SimplePool is a default implementation **IObjectPool** Spring.Aop.Target.SimplePoolTargetSource is the implementation of IPoolableObjectFactory for use with AOP. The current goal of the Spring.Pool namespace is not to provide a one-for-one replacement of the Jakarta Commons Pool API, but rather to support basic object pooling needs for common AOP scenarios. Consequently, other interfaces and base classes available in the Jakarta package are not available.

Chapter 10. Spring.NET miscellanea

10.1. Introduction

This chapter contains miscellanea information on features, goodies, caveats that does not belong to any paricular area.

10.2. PathMatcher

Note, Spring.Util.PathMatcher is currently only available in CVS, not the RC3 release. If you want to use these feature please get the code from CVS (instructions) or from the download section of the Spring.NET website that contains an .zip with the full CVS tree.

Spring.Util.PathMatcher provides Ant/NAnt-like path name matching features.

To do the match, you use the method:

```
static bool Match(string pattern, string path)
```

If you want to decide if case is important or not use the method:

```
static bool Match(string pattern, string path, bool ignoreCa
```

10.2.1. General rules

To build your pattern, you use the *, ? and ** building blocks:

- *: matches any number of non slash characters;
- ?: matches exactly 1 (one) non slash/dot character;
- **: matches any subdirectory, without taking care of the depth;

10.2.2. Matching filenames

A file name can be matched using the following notation:

```
foo?bar.*
```

matches:

```
fooAbar.txt
foo1bar.txt
foo_bar.txt
foo-bar.txt
```

does not match:

```
foo.bar.txt
foo/bar.txt
foo\bar.txt
```

The classical all files pattern:

```
* *
```

matches:

```
foo.db
.db
foo
foo.bar.db
foo.db.db
db.db
```

does not match:

```
c:/
c:/foo.db
c:/foo
c:/foo
c:/.db
c:/foo.foo.db
//server/foo
```

10.2.3. Matching subdirectories

A directory name can be matched at any depth level using the following notation:

```
**/db/**
```

That pattern matches the following paths:

```
/db
//server/db
c:/db
c:/spring/app/db/foo.db
//Program Files/App/spaced dir/db/foo.db
```

```
/home/spring/spaced dir/db/v1/foo.db
```

but does not match these:

```
c:/spring/app/db-v1/foo.db
/home/spring/spaced dir/db-v1/foo.db
```

You can compose subdirectories to match like this:

```
**/bin/**/tmp/**
```

That pattern matches the following paths:

```
c:/spring/foo/bin/bar/tmp/a
c:/spring/foo/bin/tmp/a/b.c
```

but does not match these:

```
c:/spring/foo/bin/bar/temp/a
c:/tmp/foo/bin/bar/a/b.c
```

You can use more advanced patterns:

```
**/.spring-assemblies*/**
```

matches:

```
c:/.spring-assemblies
c:/.spring-assembliesabcd73xs
c:/app/.spring-assembliesabcd73xs
c:/app/.spring-assembliesabcd73xs/foo.dll
//server/app/.spring-assembliesabcd73xs
```

does not match:

```
c:/app/.spring-assemblie
```

10.2.4. Case does matter, slashes don't

.NET is expected to be a cross-platform development ... platform. So, PathMatcher will match taking care of the case of the pattern and the case of the path. For example:

```
**/db/**/*.DB
```

matches:

```
c:/spring/service/deploy/app/db/foo.DB
```

but does not match:

```
c:/spring/service/deploy/app/DB/foo.DB
c:spring/service/deploy/app/spaced dir/DB/foo.DB
//server/share/service/deploy/app/DB/backup/foo.db
```

If you do not matter about case, you should explicitly tell the Pathmatcher. Back and forward slashes, in the very same cross-platform spirit, are not important:

```
spring/foo.bar
```

matches all the following paths:

```
c:\spring\foo.bar
c:/spring\foo.bar
c:/spring/foo.bar
/spring/foo.bar
\spring\foo.bar
```

Chapter 11. Expression Evaluation

11.1. Introduction

The Spring.Expressions namespace provides a powerful expression language for querying and manipulating an object graph at runtime. The language supports setting and getting of property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection, as well as common list aggregators.

The functionality provided in this namespace serves as the foundation for a variety of other features in Spring.NET such as enhanced property evaluation in the XML based configuration of the IoC container, a Data Validation framework, and a Data Binding framework for ASP.NET. You will likely find other cool uses for this library in your own work where run-time evaluation of criteria based on an object's state is required. For those with a Java background, the Spring.Expressions namespace provides functionality similar to the Java based Object Graph Navigation Language, OGNL.

This chapter covers the features of the expression language using an Inventor and Inventor's Society class as the target objects for expression evaluation. The class declarations and the data used to populate them are listed at the end of the chapter in section <u>Section 11.4</u>, "Classes used in the examples". These classes are blatantly taken from the NUnit tests for the Expressions namespace which you can refer to for additional example usage.

11.2. Evaluating Expressions

The simplest, but not the most efficient way to perform expression evaluation is by using one of the static convenience methods of the ExpressionEvaluator class:

```
public static object GetValue(object root, string expression
public static object GetValue(object root, string expression
public static void SetValue(object root, string expression,
public static void SetValue(object root, string expression,
```

The first argument is the 'root' object that the expression string (2nd argument) will be evaluated against. The third argument is used to support variables in the expression and will be discussed later. Simple usage to get the value of an object property is shown below using the <code>Inventor</code> class. You can find the class listing in section <u>Section 11.4, "Classes used in the examples"</u>.

```
Inventor tesla = new Inventor("Nikola Tesla", new DateTime(1
tesla.PlaceOfBirth.City = "Smiljan";
string evaluatedName = (string) ExpressionEvaluator.GetValue
string evaluatedCity = (string) ExpressionEvaluator.GetValue
```

The value of 'evaluatedName' is 'Nikola Tesla' and that of 'evaluatedCity' is 'Smiljan'. A period is used to navigate the nested properties of the object. Similarly to set the property of an object, say we want to rewrite history and change Tesla's city of birth, we would simply add the following line

```
ExpressionEvaluator.SetValue(tesla, "PlaceOfBirth.City", "No
```

A much better way to evaluate expressions is to parse them once and then evaluate as many times as you want using Expression class. Unlike Expression Evaluator, which parses expression every time you

invoke one of its methods, **Expression** class will cache the parsed expression for increased performance. The methods of this class are listed below:

```
public static IExpression Parse(string expression)
public override object Get(object context, IDictionary variable
public override void Set(object context, IDictionary variable)
```

The retrieval of the Name property in the previous example using the Expression class is shown below

```
IExpression exp = Expression.Parse("Name");
string evaluatedName = (string) exp.GetValue(tesla, null);
```

The difference in performance between the two approaches, when evaluating the same expression many times, is several orders of magnitude, so you should only use convenience methods of the <code>ExpressionEvaluator</code> class when you are doing one-off expression evaluations. In all other cases you should parse the expression first and then evaluate it as many times as you need.

There are a few exception classes to be aware of when using the ExpressionEvaluator. These are InvalidPropertyException, when you refer to a property that doesn't exist, NullValueInNestedPathException, when a null value is encountered when traversing through the nested property list, and ArgumentException and NotSupportedException when you pass in values that are in error in some other manner.

The expression language is based on a grammar and uses ANTLR to construct the lexer and parser. Errors relating to bad syntax of the language will be caught at this level of the language implementation. For those interested in the digging deeper into the implementation, the grammar file is named Expression.g and is located in the src directory of the namespace. As a side note, the release version of the ANTLR DLL included with Spring.NET was signed with the Spring.NET key, which means that you should always use the included version of antlr.runtime.dll within your application. Upcoming releases of

ANTLR will provide strongly signed assemblies, which will remove this requirement.

11.3. Language Reference

11.3.1. Literal expressions

The types of literal expressions supported are strings, dates, numeric values (int, real, and hex), boolean and null. String are delimited by single quotes. To put a single quote itself in a string use the backslash character. The following listing shows simple usage of literals. Typically they would not be used in isolation like this, but as part of a more complex expression, for example using a literal on one side of a logical comparison operator.

```
string helloworld = (string) ExpressionEvaluator.GetValue(nu
string tonyPizza = (string) ExpressionEvaluator.GetValue(nu
double avogadrosNumber = (double) ExpressionEvaluator.GetVal
int maxValue = (int) ExpressionEvaluator.GetValue(null, "0x
DateTime birthday = (DateTime) ExpressionEvaluator.GetValue(
DateTime exactBirthday =
    (DateTime) ExpressionEvaluator.GetValue(null, "date('1974)
bool trueValue = (bool) ExpressionEvaluator.GetValue(null, "object nullValue = ExpressionEvaluator.GetValue(null, "null")
```

Note that the extra backslash character in Tony's Pizza is to satisfy C# escape syntax. Numbers support the use of the negative sign, exponential notation, and decimal points. By default real numbers are parsed using Double.Parse unless the format character "M" or "F" is supplied, in which case Decimal.Parse and Single.Parse would be used respectfully. As shown above, if two arguments are given to the date literal then DateTime.ParseExact will be used. Note that all parse methods of classes that are used internally reference the CultureInfo.InvariantCulture.

11.3.2. Properties, Arrays, Lists, Dictionaries, Indexers

As shown in the previous example in <u>Section 11.2</u>, <u>"Evaluating Expressions"</u>, navigating through properties is easy, just use a period to indicate a nested property value. The instances of **Inventor** class, *pupin* and *tesla*, were populated with data listed in section <u>Section 11.4</u>, <u>"Classes used in the examples"</u>. To navigate "down" and get Tesla's year of birth and Pupin's city of birth the following expressions are used

```
int year = (int) ExpressionEvaluator.GetValue(tesla, "DOB.Ye
string city = (string) ExpressionEvaluator.GetValue(pupin, "
```

For the sharp-eyed, that isn't a typo in the property name for place of birth. The expression uses mixed cases to demonstrate that the evaluation is case insensitive.

The contents of arrays and lists are obtained using square bracket notation.

```
// Inventions Array
string invention = (string) ExpressionEvaluator.GetValue(tes
// Members List
string name = (string) ExpressionEvaluator.GetValue(ieee, "N
// List and Array navigation
string invention = (string) ExpressionEvaluator.GetValue(iee)
```

The contents of dictionaries are obtained by specifying the literal key value within the brackets. In this case, because keys for the *Officers* dictionary are strings, we can specify string literal.

```
// Officer's Dictionary
Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(iee
string city = (string) ExpressionEvaluator.GetValue(ieee, "C
ExpressionEvaluator.SetValue(ieee, "Officers['advisors'][0].
```

You may also specify non literal values in place of the quoted literal values by using another expression inside the square brackets such as variable names or static properties/methods on other types. These features are discussed in other

sections.

Indexers are similarly referenced using square brackets. The following is a small example that shows the use of indexers. Multidimensional indexers are also supported.

```
public class Bar
{
    private int[] numbers = new int[] {1, 2, 3};

    public int this[int index]
    {
        get { return numbers[index];}
        set { numbers[index] = value; }
    }
}

Bar b = new Bar();
int val = (int) ExpressionEvaluator.GetValue(bar, "[1]") //

ExpressionEvaluator.SetValue(bar, "[1]", 3); // set value to the content of the content
```

11.3.2.1. Defining Arrays, Lists and Dictionaries Inline

In addition to accessing arrays, lists and dictionaries by navigating the graph for the context object, Spring.NET Expression Language allows you to define them inline, within the expression. Inline lists are defined by simply enclosing a comma separated list of items with curly brackets:

```
{1, 2, 3, 4, 5}
{'abc', 'xyz'}
```

If you want to ensure that a strongly typed array is initialized instead of a weakly typed list, you can use array initializer instead:

```
new int[] {1, 2, 3, 4, 5}
new string[] {'abc', 'xyz'}
```

Dictionary definition syntax is a bit different: you need to use a # prefix to tell

expression parser to expect key/value pairs within the brackets and to specify a comma separated list of key/value pairs within the brackets:

```
#{'key1' : 'Value 1', 'today' : DateTime.Today}
#{1 : 'January', 2 : 'February', 3 : 'March', ...}
```

Arrays, lists and dictionaries created this way can be used anywhere where arrays, lists and dictionaries obtained from the object graph can be used, which we will see later in the examples.

Keep in mind that even though examples above use literals as array/list elements and dictionary keys and values, that's only to simplify the examples -- you can use any valid expression wherever literals are used.

11.3.3. Methods

Methods are invoked using typical C# programming syntax. You may also invoke methods on literals.

```
//string literal
char[] chars = (char[]) ExpressionEvaluator.GetValue(null, '
//date literal
int year = (int) ExpressionEvaluator.GetValue(null, "date('1
// object usage, calculate age of tesla navigating from the
ExpressionEvaluator.GetValue(ieee, "Members[0].GetAge(date('))
```

11.3.4. Operators

11.3.4.1. Relational operators

The relational operators; equal, not equal, less than, less than or equal, greater than, and greater than or equal are supported using standard operator notation. These operators take into account if the object implements the <code>IComparable</code> interface. Enumerations are also supported but you will need to register the enumeration type, as described in Section Section 11.3.8, "Type Registration", in order to use an enumeration value in an expression if it is not contained in the mscorlib.

Enumerations can be evaluated as shown below

```
FooColor fColor = new FooColor();

ExpressionEvaluator.SetValue(fColor, "Color", KnownColor.Blu
bool trueValue = (bool) ExpressionEvaluator.GetValue(fColor,
```

Where FooColor is the following class.

```
public class FooColor
{
    private KnownColor knownColor;

    public KnownColor Color
    {
       get { return knownColor;}
       set { knownColor = value; }
    }
}
```

In addition to standard relational operators, Spring.NET Expression Language supports some additional, very useful operators that were "borrowed" from SQL, such as *in*, *like* and *between*, as well as *is* and *matches* operators, which allow you to test if object is of a specific type or if the value matches a regular expression.

```
ExpressionEvaluator.GetValue(null, "3 in {1, 2, 3, 4, 5}")
ExpressionEvaluator.GetValue(null, "'Abc' like '[A-Z]b*'")
```

```
ExpressionEvaluator.GetValue(null, "'Abc' like '?'") // fall ExpressionEvaluator.GetValue(null, "1 between {1, 5}") // t ExpressionEvaluator.GetValue(null, "'efg' between {'abc', 'x ExpressionEvaluator.GetValue(null, "'xyz' is int") // false ExpressionEvaluator.GetValue(null, "{1, 2, 3, 4, 5} is IList ExpressionEvaluator.GetValue(null, "'5.0067' matches '^-?\\c
ExpressionEvaluator.GetValue(null, @"'5.00' matches '^-?\\c
```

Note that the Visual Basic and not SQL syntax is used for the *like* operator pattern string.

11.3.4.2. Logical operators

The logical operators that are supported are *and*, *or*, and *not*. Their use is demonstrated below

```
// AND
bool falseValue = (bool) ExpressionEvaluator.GetValue(null,
string expression = @"IsMember('Nikola Tesla') and IsMember(
bool trueValue = (bool) ExpressionEvaluator.GetValue(ieee, 6

// OR
bool trueValue = (bool) ExpressionEvaluator.GetValue(null, '
string expression = @"IsMember('Nikola Tesla') or IsMember('
bool trueValue = (bool) ExpressionEvaluator.GetValue(ieee, 6

// NOT
bool falseValue = (bool) ExpressionEvaluator.GetValue(null,
// AND and NOT
string expression = @"IsMember('Nikola Tesla') and !IsMember
bool falseValue = (bool) ExpressionEvaluator.GetValue(ieee,
```

11.3.4.3. Mathematical operators

The addition operator can be used on numbers, strings and dates. Subtraction can be used on numbers and dates. Multiplication and division can be used only on numbers. Other mathematical operators supported are modulus (%) and exponential power (^). Standard operator precedence is enforced. These operators are demonstrated below

```
// Addition
int two = (int)ExpressionEvaluator.GetValue(null, "1 + 1");
String testString = (String)ExpressionEvaluator.GetValue(nul
DateTime dt = (DateTime)ExpressionEvaluator.GetValue(null, '
// Subtraction
int four = (int) ExpressionEvaluator.GetValue(null, "1 - -3")
Decimal dec = (Decimal) ExpressionEvaluator.GetValue(null, '
TimeSpan ts = (TimeSpan) ExpressionEvaluator.GetValue(null,
// Multiplication
int six = (int) ExpressionEvaluator.GetValue(null, "-2 * -3")
int twentyFour = (int) ExpressionEvaluator.GetValue(null, "2
// Division
int minusTwo = (int) ExpressionEvaluator.GetValue(null, "6 /
int one = (int) ExpressionEvaluator.GetValue(null, "8.0 / 46
// Modulus
int three = (int) ExpressionEvaluator.GetValue(null, "7 % 4")
int one = (int) ExpressionEvaluator.GetValue(null, "8.0 % 56
// Exponent
```

```
int sixteen = (int) ExpressionEvaluator.GetValue(null, "-2 /
// Operator precedence
int minusFortyFive = (int) ExpressionEvaluator.GetValue(null)
```

11.3.5. Assignment

Setting of a property is done by using the assignment operator. This would typically be done within a call to <code>GetValue</code> since in the simple case <code>SetValue</code> offers the same functionality. Assignment in this manner is useful when combining multiple operators in an expression list, discussed in the next section. Some examples of assignment are shown below

```
Inventor inventor = new Inventor();
String aleks = (String) ExpressionEvaluator.GetValue(inventor)
DateTime dt = (DateTime) ExpressionEvaluator.GetValue(inventor)
//Set the vice president of the society
Inventor tesla = (Inventor) ExpressionEvaluator.GetValue(iee)
```

11.3.6. Expression lists

Multiple expressions can be evaluated against the same context object by separating them with a semicolon and enclosing the entire expression within parentheses. The value returned is the value of the last expression in the list. Examples of this are shown below

```
//Perform property assignments and then return Name property
String pupin = (String) ExpressionEvaluator.GetValue(ieee.Me
   "( [1].PlaceOfBirth.City = 'Beograd'; [1].PlaceOfBirth.Cot
// pupin = "Mihajlo Pupin"
```

11.3.7. Types

In many cases, you can reference types by simply specifying type name:

```
ExpressionEvaluator.GetValue(null, "1 is int")
ExpressionEvaluator.GetValue(null, "DateTime.Today")
ExpressionEvaluator.GetValue(null, "new string[] { 'abc', 'ef
```

This is possible for all standard types from mscorlib, as well as for any other type that is registered with the TypeReqistry as described in the next section.

For all other types, you need to use special **T(typeName)** expression:

```
Type dateType = (Type) ExpressionEvaluator.GetValue(null,
Type evalType = (Type) ExpressionEvaluator.GetValue(null,
bool trueValue = (bool) ExpressionEvaluator.GetValue(tesla,
```



The implementation delegates to Spring's ObjectUtils. ResolveType method for the actual type resolution, which means that the types used within expressions are resolved in the exactly the same way as the types specified in Spring configuration files.

11.3.8. Type Registration

To refer to a type within an expression that is not in the mscorlib you need to register it with the TypeReqistry. This will allow you to refer to a shorthand name of the type within your expressions. This is commonly used in expression that use the new operator or refer to a static properties of an object. Example usage is shown below.

```
TypeRegistry.RegisterType("Society", typeof(Society));
Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(iee
```

Alternatively, you can register types using typeAliases configuration section.

11.3.9. Constructors

Constructors can be invoked using the new operator. For classes outside mscorlib you will need to register your types so they can be resolved. Examples of using constructors are shown below:

```
// simple ctor
DateTime dt = (DateTime) ExpressionEvaluator.GetValue(null,

// Register Inventor type then create new inventor instance
// Then return the new count of the Members collection.

TypeRegistry.RegisterType(typeof(Inventor));
int three = (int) ExpressionEvaluator.GetValue(ieee.Members,
```

As a convenience, Spring.NET also allows you to define named constructor arguments, which are used to set object's properties after instantiation, similar to the way standard .NET attributes work. For example, you could create an instance of the <code>Inventor</code> class and set its <code>Inventions</code> property in a single statement:

```
Inventor aleks = (Inventor) ExpressionEvaluator.GetValue(nul
```

The only rule you have to follow is that named arguments should be specified *after* standard constructor arguments, just like in the .NET attributes.

While we are on the subject, Spring.NET Expression Language also provides a convenient syntax for .NET attribute instance creation. Instead of using standard constructor syntax, you can use a somewhat shorter and more familiar syntax to create an instance of a .NET attribute class:

```
WebMethodAttribute webMethod = (WebMethodAttribute) Expressi
```

As you can see, with the exception of the @ prefix, syntax is exactly the same as in C#.

Slightly different syntax is not the only thing that differentiates an attribute expression from a standard constructor invocation expression. In addition to that, attribute expression uses slightly different type resolution mechanism and will

attempt to load both the specified type name and the specified type name with an Attribute suffix, just like the C# compiler.

11.3.10. Variables

Variables can referenced in the expression using the syntax #variableName. The variables are passed in and out of the expression using the dictionary parameter in ExpressionEvaluator's GetValue or SetValue methods.

```
public static object GetValue(object root, string expression
public static void SetValue(object root, string expression,
```

The variable name is the key value of the dictionary. Example usage is shown below;

```
IDictionary vars = new Hashtable();
vars["newName"] = "Mike Tesla";
ExpressionEvaluator.GetValue(tesla, "Name = #newName", vars)
```

You can also use the dictionary as a place to store values of the object as they are evaluated inside the expression. For example to change Tesla's first name back again and keep the old value;

```
ExpressionEvaluator.GetValue(tesla, "{ #oldName = Name; Name
String oldName = (String)vars["oldName"]; // Mike Tesla
```

Variable names can also be used inside indexers or maps instead of literal values. For example;

```
vars["prez"] = "president";
Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(iee
```

11.3.10.1. The '#this' and '#root' variables

There are two special variables that are always defined and can be references within the expression: #this and #root.

The #this variable can be used to explicitly refer to the context for the node

that is currently being evaluated:

```
// sets the name of the president and returns its instance
ExpressionEvaluator.GetValue(ieee, "Officers['president'].(
```

Similarly, the **#root** variable allows you to refer to the root context for the expression:

```
// removes president from the Officers dictionary and return
ExpressionEvaluator.GetValue(ieee, "Officers['president'].(
```

11.3.11. Ternary Operator (If-Then-Else)

You can use the ternary operator for performing if-then-else conditional logic inside the expression. A minimal example is;

```
String aTrueString = (String) ExpressionEvaluator.GetValue(
```

In this case, the boolean false results in returning the string value 'trueExp'. A less artificial example is shown below

11.3.12. List Projection and Selection

List projection and selection are very powerful expression language features that allow you to transform the source list into another list by either *projecting* across its "columns", or *selecting* from its "rows". In other words, projection can be thought of as a column selector in a SQL SELECT statement, while selection

would be comparable to the WHERE clause.

For example, let's say that we need a list of the cities where our inventors were born. This could be easily obtained by projecting on the PlaceOfBirth.City property:

```
IList placesOfBirth = (IList) ExpressionEvaluator.GetValue(i
```

Or we can get the list of officers' names:

```
IList officersNames = (IList) ExpressionEvaluator.GetValue(i
```

As you can see from the examples, projection uses ! {projectionExpression} syntax and will return a new list of the same length as the original list but typically with the elements of a different type.

On the other hand, selection, which uses **?**{*projectionExpression*} syntax, will filter the list and return a new list containing a subset of the original element list. For example, selection would allow us to easily get a list of Serbian inventors:

```
IList serbianInventors = (IList) ExpressionEvaluator.GetValu
```

Or to get a list of inventors that invented sonar:

```
IList sonarInventors = (IList) ExpressionEvaluator.GetValue(
```

Or we can combine selection and projection to get a list of sonar inventors' names:

```
IList sonarInventorsNames = (IList) ExpressionEvaluator.GetV
```

As a convenience, Spring.NET Expression Language also supports a special syntax for selecting the first or last match. Unlike regular selection, which will return an empty list if no matches are found, first or last match selection expression will either return an instance of the matched element, or null if no matching elements were found. In order to return a first match you should prefix your selection expression with f instead of f, and to return last match you should use f prefix:

```
ExpressionEvaluator.GetValue(ieee, "Members.^{Nationality ==
ExpressionEvaluator.GetValue(ieee, "Members.${Nationality ==}
```

Notice that we access the **Name** property directly on the selection result, because an actual matched instance is returned by the first and last match expression instead of a filtered list.

11.3.13. Collection Processors and Aggregators

In addition to list projection and selection, Spring.NET Expression Language also supports several collection processors, such as distinct, nonNull and SOrt, as well as a number of commonly used aggregators, such as Max, min, count, sum and average.

The difference between processors and aggregators is that processors return a new or transformed collection, while aggregators return a single value. Other than that, they are very similar -- both processors and aggregators are invoked on a collection node using standard method invocation expression syntax, which makes them very simple to use and allows easy chaining of multiple processors.

11.3.13.1. Count Aggregator

The count aggregator is a safe way to obtain a number of items in a collection. It can be applied to a collection of any type, including arrays, which helps eliminate the decision on whether to use Count or Length property depending on the context. Unlike its standard .NET counterparts, count aggregator can also be invoked on the null context without throwing a NullReferenceException. It will simply return zero in this case, which makes it much safer than standard .NET properties within larger expression.

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3}.count()") //
ExpressionEvaluator.GetValue(null, "count()") // 0
```

11.3.13.2. Sum Aggregator

The sum aggregator can be used to calculate a total for the list of numeric values. If numbers within the list are not of the same type or precision, it will automatically perform necessary conversion and the result will be the highest

precision type. If any of the collection elements is not a number, this aggregator will throw an InvalidArgumentException.

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3, 10}.sum()")
ExpressionEvaluator.GetValue(null, "{5, 5.8, 12.2, 1}.sum()"
```

11.3.13.3. Average Aggregator

The average aggregator will return the average for the collection of numbers. It will use the same type coercion rules, as the sum aggregator in order to be as precise as possible. Just like the sum aggregator, if any of the collection elements is not a number, it will throw an InvalidArgumentException.

```
ExpressionEvaluator.GetValue(null, "{1, 5, -4, 10}.average()
ExpressionEvaluator.GetValue(null, "{1, 5, -2, 10}.average()
```

11.3.13.4. Minimum Aggregator

The minimum aggregator will return the smallest item in the list. In order to determine what "the smallest" actually means, this aggregator relies on the assumption that the collection items are of the uniform type and that they implement the <code>IComparable</code> interface. If that is not the case, this aggregator will throw an <code>InvalidArgumentException</code>.

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3, 10}.min()")
ExpressionEvaluator.GetValue(null, "{'abc', 'efg', 'xyz'}.mi
```

11.3.13.5. Maximum Aggregator

The maximum aggregator will return the largest item in the list. In order to determine what "the largest" actually means, this aggregator relies on the assumption that the collection items are of the uniform type and that they implement <code>IComparable</code> interface. If that is not the case, this aggregator will throw an <code>InvalidArgumentException</code>.

```
ExpressionEvaluator.GetValue(null, "{1, 5, -3, 10}.max()")
ExpressionEvaluator.GetValue(null, "{'abc', 'efg', 'xyz'}.ma
```

11.3.13.6. Non-null Processor

A non-null processor is a very simple collection processor that eliminates all **null** values from the collection.

```
ExpressionEvaluator.GetValue(null, "{ 'abc', 'xyz', null, 'a
ExpressionEvaluator.GetValue(null, "{ 'abc', 'xyz', null, 'a
```

11.3.13.7. Distinct Processor

A distinct processor is very useful when you want to ensure that you don't have duplicate items in the collection. It can also accept an optional Boolean argument that will determine whether null values should be included in the results. The default is false, which means that they will not be included.

```
ExpressionEvaluator.GetValue(null, "{ 'abc', 'xyz', 'abc', '
ExpressionEvaluator.GetValue(null, "{ 'abc', 'xyz', 'abc', '
```

11.3.13.8. Sort Processor

The sort processor can be used to sort uniform collections of elements that implement **IComparable**.

```
ExpressionEvaluator.GetValue(null, "{1.2, 5.5, -3.3}.sort()'
ExpressionEvaluator.GetValue(null, "{ 'abc', 'xyz', 'abc', '
```

The sort processor also accepts a boolean value as an argument to determine sort order, sort(false) will sort the collection in decending order.

11.3.13.9. Type Conversion Processor

The convert processor can be used to convert a collection of elements to a given Type.

```
object[] arr = new object[] { "0", 1, 1.1m, "1.1", 1.1f };
decimal[] result = (decimal[]) ExpressionEvaluator.GetValue(
```

11.3.13.10. Reverse Processor

The reverse processor returns the reverse order of elements in the list

```
object[] arr = new object[] { "0", 1, 2.1m, "3", 4.1f };
object[] result = new ArrayList( (ICollection) ExpressionEva
```

11.3.13.11. OrderBy Processor

Collections can be ordered in three ways, an expression, a SpEL lamda expreression, or a delegate.

```
// orderBy expression
IExpression exp = Expression.Parse("orderBy('ToString()')");
object[] input = new object[] { 'b', 1, 2.0, "a" };
object[] ordered = exp.GetValue(input); // { 1, 2.0, "a",
// SpEL lambda expressions
IExpression exp = Expression.Parse("orderBy({|a,b| $a.ToStri
object[] input = new object[] { 'b', 1, 2.0, "a" };
object[] ordered = exp.GetValue(input); // { 1, 2.0, "a",
Hashtable vars = new Hashtable();
Expression.RegisterFunction( "compare", "{|a,b| $a.ToString(
exp = Expression.Parse("orderBy(#compare)");
ordered = exp.GetValue(input, vars); // { 1, 2.0, "a", 'b'
// .NET delegate
private delegate int CompareCallback(object x, object y);
private int CompareObjects(object x, object y)
{
  if (x == y) return 0;
 return x.ToString().CompareTo(""+y);
}
Hashtable vars = new Hashtable();
vars["compare"] = new CompareCallback(CompareObjects);
IExpression exp = Expression.Parse("orderBy(#compare)");
object[] input = new object[] { 'b', 1, 2.0, "a" };
object[] ordered = exp.GetValue(input); // { 1, 2.0, "a",
```

11.3.13.12. User Defined Collection Processor

You can register your own collection processor for use in evaluation a collection. Here is an example of a ICollectionProcessor implementation that sums only the even numbers of an integer list

```
public class IntEvenSumCollectionProcessor : IColled
    public object Process(ICollection source, object
        object total = 0d;
        foreach (object item in source)
            if (item != null)
            {
                if (NumberUtils.IsInteger(item))
                {
                    if ((int)item % 2 == 0)
                         total = NumberUtils.Add(tota
                }
                else
                {
                    throw new ArgumentException("Sum
                }
            }
        }
        return total;
    }
}
public void DoWork()
    Hashtable vars = new Hashtable();
    vars["EvenSum"] = new IntEvenSumCollectionProces
    int result = (int)ExpressionEvaluator.GetValue(r
}
```

11.3.14. Spring Object References

Expressions can refer to objects that are declared in Spring's application context using the syntax <code>@(contextName:objectName)</code>. If no contextName is specified the default root context name (<code>Spring.RootContext</code>) is used. Using the application context defined in the MovieFinder example from <code>Chapter 33, IoC Quickstarts</code>, the following expression returns the number of movies directed by Roberto Benigni.

The variable numMovies is evaluated to 2 in this example.

11.3.15. Lambda Expressions

A somewhat advanced, but a very powerful feature of Spring.NET Expression Language are lambda expressions. Lambda expressions allow you to define inline functions, which can then be used within your expressions just like any other function or method. You may also use .NET delegates as described in the next section.

The syntax for defining lambda expressions is:

```
#functionName = { | argList | functionBody }
```

For example, you could define a **max** function and call it like this:

```
ExpressionEvaluator.GetValue(null, "(\#max = {|x,y| \$x > \$y ?
```

As you can see, any arguments defined for the expression can be referenced within the function body using a *local variable* syntax, \$*varName*. Invocation of the function defined using lambda expression is as simple as specifying the comma-separated list of function arguments in parentheses, after the function name.

Lambda expressions can be recursive, which means that you can invoke the function within its own body:

Notice that in both examples above we had to specify a Variables parameter for the GetValue method. This is because lambda expressions are actually nothing more than parameterized variables and we need variables dictionary in order to store them. If you don't specify a valid IDictionary instance for the Variables parameter, you will get a runtime exception.

Also, in both examples above we used an expression list in order to define and invoke a function in a single expression. However, more likely than not, you will want to define your functions once and then use them within as many expressions as you need. Spring.NET provides an easy way to pre-register your lambda expressions by exposing a static <code>Expression.RegisterFunction</code> method, which takes function name, lambda expression and variables dictionary to register function in as parameters:

```
IDictionary vars = new Hashtable();
Expression.RegisterFunction("sqrt", "{|n| Math.Sqrt($n)}", v
Expression.RegisterFunction("fact", "{|n| $n <= 1 ? 1 : $n *</pre>
```

Once the function registration is done, you can simply evaluate an expression that uses these functions, making sure that the **Vars** dictionary is passed as a parameter to expression evaluation engine:

```
ExpressionEvaluator.GetValue(null, "#fact(5)", vars) // 126
ExpressionEvaluator.GetValue(null, "#sqrt(9)", vars) // 3
```

Finally, because lambda expressions are treated as variables, they can be

assigned to other variables or passed as parameters to other lambda expressions. In the following example we are defining a delegate function that accepts function f as the first argument and parameter n that will be passed to function f as the second. Then we invoke the functions registered in the previous example, as well as the lambda expression defined inline, through our delegate:

```
Expression.RegisterFunction("delegate", "{|f, n| $f($n) }", ExpressionEvaluator.GetValue(null, "#delegate(#sqrt, 4)", value(null, "#delegate(#fact, 5)", value(null, "#delegate(#fact, 5)", value(null, "#delegate({|n| $n ^ 2 },
```

While this particular example is not particularly useful, it does demonstrate that lambda expressions are indeed treated as nothing more than parameterized variables, which is important to remember.

11.3.16. Delegate Expressions

Delegate expressions allow you to refer to .NET delegates which can then be used within your expressions just like any other function or method.

For example, you can define a max delegate and call it like this

```
private delegate double DoubleFunctionTwoArgs(double arg1, double arg2)
{
   return Math.Max(arg1, arg2);
}

public void DoWork()
{
   Hashtable vars = new Hashtable();
   vars["max"] = new DoubleFunctionTwoArgs(Max);
   double result = (double) ExpressionEvaluator.GetValue(null)
}
```

11.3.17. Null Context

If you do not specify a root object, i.e. pass in null, then the expressions evaluated either have to be literal values, i.e. ExpressionEvaluator.GetValue(null,

"2 + 3.14"), refer to classes that have static methods or properties, i.e. ExpressionEvaluator.GetValue(null, "DateTime.Today"), create new instances of objects, i.e. ExpressionEvaluator.GetValue(null, "new DateTime(2004, 8, 14)") or refer to other objects such as those in the variable dictionary or in the IoC container. The latter two usages will be discussed later.

11.4. Classes used in the examples

The following simple classes are used to demonstrate the functionality of the expression language.

```
public class Inventor
{
    public string Name;
    public string Nationality;
    public string[] Inventions;
    private DateTime dob;
    private Place pob;
    public Inventor() : this(null, DateTime.MinValue, null)
    {}
    public Inventor(string name, DateTime dateOfBirth, string
        this.Name = name;
        this.dob = dateOfBirth;
        this. Nationality = nationality;
        this.pob = new Place();
    }
    public DateTime DOB
    {
        get { return dob; }
        set { dob = value; }
    }
    public Place PlaceOfBirth
        get { return pob; }
    }
    public int GetAge(DateTime on)
    {
        // not very accurate, but it will do the job ;-)
        return on.Year - dob.Year;
}
```

```
public class Place
    public string City;
    public string Country;
}
public class Society
    public string Name;
    public static string Advisors = "advisors";
    public static string President = "president";
    private IList members = new ArrayList();
    private IDictionary officers = new Hashtable();
    public IList Members
    {
        get { return members; }
    }
    public IDictionary Officers
        get { return officers; }
    }
    public bool IsMember(string name)
        bool found = false;
        foreach (Inventor inventor in members)
            if (inventor.Name == name)
            {
                found = true;
                break;
        return found;
    }
}
```

The code listings in this chapter use instances of the data populated with the following information.

```
Inventor tesla = new Inventor("Nikola Tesla", new DateTime(1
tesla.Inventions = new string[]
    {
        "Telephone repeater", "Rotating magnetic field princ
        "Polyphase alternating-current system", "Induction m
        "Alternating-current power transmission", "Tesla coi
        "Wireless communication", "Radio", "Fluorescent ligh
    };
tesla.PlaceOfBirth.City = "Smiljan";
Inventor pupin = new Inventor("Mihajlo Pupin", new DateTime(
pupin.Inventions = new string[] {"Long distance telephony &
pupin.PlaceOfBirth.City = "Idvor";
pupin.PlaceOfBirth.Country = "Serbia";
Society ieee = new Society();
ieee.Members.Add(tesla);
ieee.Members.Add(pupin);
ieee.Officers["president"] = pupin;
ieee.Officers["advisors"] = new Inventor[] {tesla, pupin};
```

Chapter 12. Validation Framework

12.1. Introduction

Data validation is a very important part of any enterprise application. ASP.NET has a validation framework but it is very limited in scope and starts falling apart as soon as you need to perform more complex validations. Problems with the out of the box ASP.NET validation framework are well documented by Peter Blum on his web site, so we are not going to repeat them here. Peter has also built a nice replacement for the standard ASP.NET validation framework, which is worth looking into if you prefer the standard ASP.NET validation mechanism to the one offered by Spring.NET for some reason. Both frameworks will allow you to perform very complex validations but we designed the Spring.NET validation framework differently for the reasons described below.

On the Windows Forms side the situation is even worse. Out of the box data validation features are completely inadequate as pointed out by Ian Griffiths in this <u>article</u>. One of the major problems we saw in most validation frameworks available today, both open source and commercial, is that they are tied to a specific presentation technology. The ASP.NET validation framework uses ASP.NET controls to define validation rules, so these rules end up in the HTML markup of your pages. Peter Blum's framework uses the same approach. In our opinion, validation is not applicable only to the presentation layer so there is no reason to tie it to any particular technology. As such, the Spring.NET Validation Framework is designed in a way that enables data validation in different application layers using the same validation rules.

The goals of the validation framework are the following:

- 1. Allow for the validation of any object, whether it is a UI control or a domain object.
- 2. Allow the same validation framework to be used in both Windows Forms and ASP.NET applications, as well as in the service layer (to validate parameters passed to the service, for example).
- 3. Allow composition of the validation rules so arbitrarily complex validation rule sets can be constructed.
- 4. Allow validators to be conditional so they only execute if a specific condition is met.

The following sections will describe in more detail how these goals were achieved and show you how to use the Spring.NET Validation Framework in

your applications.

12.2. Example Usage

Decoupling validation from presentation was the major goal that significantly influenced design of the validation framework. We wanted to be able to define a set of validation rules that are completely independent from the presentation so we can reuse them (or at least have the ability to reuse them) in different application layers. This meant that the approach taken by Microsoft ASP.NET team would not work and custom validation controls were not an option. The approach taken was to configure validation rules just like any other object managed by Spring - within the application context. However, due to possible complexity of the validation rules we decided not to use the standard Spring.NET configuration schema for validator definitions but to instead provide a more specific and easier to use custom configuration schema for validation. Note that the validation framework is not tied to the use of XML, you can use its API Programatically. The following example shows validation rules defined for the Trip object in the SpringAir sample application:

```
<objects xmlns="http://www.springframework.net" xmlns:v="htt</pre>
  <object type="TripForm.aspx" parent="standardPage">
     cproperty name="TripValidator" ref="tripValidator" />
  </object>
  <v:group id="tripValidator">
     <v:required id="departureAirportValidator" test="Starti</pre>
        <v:message id="error.departureAirport.required" prov</pre>
     </v:required>
     <v:group id="destinationAirportValidator">
        <v:required test="ReturningFrom.AirportCode">
           <v:message id="error.destinationAirport.required"</pre>
        </v:required>
        <v:condition test="ReturningFrom.AirportCode != Star</pre>
           <v:message id="error.destinationAirport.sameAsDer</pre>
        </v:condition>
     </v:group>
     <v:group id="departureDateValidator">
        <v:required test="StartingFrom.Date">
```

```
<v:message id="error.departureDate.required" prov</pre>
        </v:required>
        <v:condition test="StartingFrom.Date >= DateTime.Tod
           <v:message id="error.departureDate.inThePast" pro</pre>
        </v:condition>
     </v:group>
     <v:group id="returnDateValidator" when="Mode == 'RoundT</pre>
        <v:required test="ReturningFrom.Date">
           <v:message id="error.returnDate.reguired" provide
        </v:required>
        <v:condition test="ReturningFrom.Date >= StartingFrd
           <v:message id="error.returnDate.beforeDeparture"</pre>
        </v:condition>
     </v:group>
  </v:group>
</objects>
```

There are a few things to note in the example above:

- You need to reference the validation schema by adding a xmlns:v="http://www.springframework.net/validation to the root element.
- You can mix standard object definitions and validator definitions in the same configuration file as long as both schemas are referenced.
- The Validator defined in the configuration file is identified by and id attribute and can be referenced in the standard Spring way, i.e. the injection of tripValidator into TripForm.aspx page definition in the first <object> tag above.
- The validation framework uses Spring's powerful expression evaluation engine to evaluate both validation rules and applicability conditions for the validator. As such, any valid Spring expression can be specified within the test and when attributes of any validator.

The example above shows many of the features of the framework, so let's discuss them one by one in the following sections.

12.3. Validator Groups

Validators can be grouped together. This is important for many reasons but the most typical usage scenario is to group multiple validation rules that apply to the same value. In the example above there is a validator group for almost every property of the Trip instance. There is also a top-level group for the Trip object itself that groups all other validators.

There are three types of validator groups each with a different behavior:

While the first type (AND) is definitely the most useful, the other two allow you to implement some specific validation scenarios in a very simple way, so you should keep them in mind when designing your validation rules.

Table 12.1. Validator Groups

Type	XML Tag	Behavior
AND	group	Returns <i>true</i> only if all contained validators return <i>true</i> . This is the most commonly used validator group.
OR	any	Returns <i>true</i> if one or more of the contained validators return <i>true</i> .
XOR	exclusive	Returns true if only one of the contained validators return true.

One thing to remember is that a validator group is a validator like any other and can be used anywhere validator is expected. You can nest groups within other groups and reference them using validator reference syntax (described later), so they really allow you to structure your validation rules in the most reusable way.

12.4. Validators

Ultimately, you will have one or more validator definitions for each piece of data that you want to validate. Spring.NET has several built-in validators that are sufficient for most validations, even fairly complex ones. The framework is extensible so you can write your own custom validators and use them in the same way as the built-in ones.

12.4.1. Condition Validator

The condition validator evaluates any logical expression that is supported by Spring's evaluation engine. The syntax is

```
<v:condition id="id" test="testCondition" when="applicabilit</pre>
  actions
</v:condition>
```

An example is shown below

```
<v:condition test="StartingFrom.Date >= DateTime.Today" wher
   <v:message id="error.departureDate.inThePast" providers='</pre>
</v:condition>
```

In this example the StartingFrom property of the Trip object is compared to see if it is later than the current date, i.e. DateTime but only when the date has been set (the initial value of StartingFrom.Date was set to DateTime.MinValue).

The condition validator could be considered "the mother of all validators". You can use it to achieve almost anything that can be achieved by using other validator types, but in some cases the test expression might be very complex, which is why you should use more specific validator type if possible. However, condition validator is still your best bet if you need to check whether particular value belongs to a particular range, or perform a similar test, as those conditions are fairly easy to write.



Keep in mind that Spring.NET Validation Framework typically works with domain objects. This is after data binding from the controls has been performed so that the object being validated is strongly typed. This means that you can easily compare numbers and dates without having to worry if the string representation is comparable.

12.4.2. Required Validator

This validator ensures that the specified test value is not empty. The syntax is

```
<v:required id="id" test="requiredValue" when="applicability
  actions
</v:required>
```

An example is shown below

```
<v:required test="ReturningFrom.AirportCode">
      <v:message id="error.destinationAirport.required" provide
</v:required>
```

The specific tests done to determine if the required value is set is listed below

Table 12.2. Rules to determine if required value is valid

System.Type	Test
System.Type	Type exists
System.String	not null or an empty string
system.DateTime	Not System.DateTime.MinValue and not system.DateTime.MaxValue
One of the number types.	not zero
System.Char	Not System.Char.MinValue or whitespace.
Any reference type other than System.String	not null

Required validator is also one of the most commonly used ones, and it is much more powerful than the ASP.NET Required validator, because it works with many other data types other than strings. For example, it will allow you to validate <code>DateTime</code> instances (both <code>MinValue</code> and <code>MaxValue</code> return <code>false</code>), integer and decimal numbers, as well as any reference type, in which case it returns <code>true</code> for a non-null value and <code>false</code> for <code>{{null}}</code>s.

The test attribute for the required validator will typically specify an expression that resolves to a property of a domain object, but it could be any valid expression that returns a value, including a method call.

12.4.3. Regular Expression Validator

The syntax is

```
<v:regex id="id" test="valueToEvaluate" when="applicabilityC
    <v:property name="Expression" value="regularExpressionToMa
    <v:property name="Options" value="regexOptions"/>
    actions
</v:regex>
```

An example is shown below

```
<v:regex test="ReturningFrom.AirportCode">
        <v:property name="Expression" value="[A-Z][A-Z][A-Z]"/>
        <v:message id="error.destinationAirport.threeCharacters"
</v:regex>
```

Regular expression validator is very useful when validating values that need to conform to some predefined format, such as telephone numbers, email addresses, URLs, etc.

One major difference of the regular expression validator compared to other builtin validator types is that you need to set a required Expression property to a regular expression to match against.

12.4.4. Generic Validator

The syntax is

```
<v:validator id="id" test="requiredValue" when="applicabilit
  actions
</v:validator>
```

An example is shown below

Generic validator allows you to plug in your custom validator by specifying its type name. Custom validators are very simple to implement, because all you need to do is extend BaseValidator class and implement abstract bool Validate(object objectToValidate) method. Your implementation simply needs to return true if it determines that object is valid, or false otherwise

12.4.5. Conditional Validator Execution

As you can see from the examples above, each validator (and validator group) allows you to define its applicability condition by specifying a logical expression as the value of the when attribute. This feature is very useful and is one of the major deficiencies in the standard ASP.NET validation framework, because in many cases specific validators need to be turned on or off based on the values of the object being validated.

For example, when validating a Trip object we need to validate return date only if the Trip.Mode property is set to the TripMode.RoundTrip enum value. In order to achieve that we created following validator definition:

```
<v:group id="returnDateValidator" when="Mode == 'RoundTrip''
    // nested validators
</v:group>
```

Validators within this group will only be evaluated for round trips.

Note Note

You should also note that you can compare enums using the string value of the enumeration. You can also use fully qualified enum name, such as:

Mode == TripMode.RoundTrip

However, in this case you need to make sure that alias for the TripMode enum type is registered using Spring's standard type aliasing mechanism.

12.5. Validator Actions

Validation actions are executed every time the containing validator is executed. They allow you to do anything you want based on the result of the validation. By far the most common use of the validation action is to add validation error message to the errors collection, but theoretically you could do anything you want. Because adding validation error messages to the errors collection is such a common scenario, Spring.NET validation schema defines a separate XML tag for this type of validation action.

12.5.1. Error Message Action

The syntax is

An example is shown below

There are several things that you have to be aware of when dealing with error messages:

- id is used to look up the error message in the appropriate Spring.NET message source.
- providers specifies a comma separated list of "error buckets" particular error message should be added to. These "buckets" will later be used by the particular presentation technology in order to display error messages as necessary.
- a message can have zero or more parameters. Each parameter is an expression that will be resolved using current validation context and the resolved values will be passed as parameters to IMessageSource.GetMessage method, which will return the

fully resolved message.

12.5.2. Generic Actions

The syntax is

```
<v:action type="actionType" when="actionApplicabilityConditi
  properties
</v:action>
```

An example is shown below

Generic actions can be used to perform all kinds of validation actions. In simple cases, such as in the example above where we turn control's visibility on or off depending on the validation result, you can use the built-in <code>ExpressionAction</code> class and simply specify expressions to be evaluated based on the validator result.

In other situations you may want to create your own action implementation, which is fairly simple thing to do — all you need to do is implement **IValidationAction** interface:

```
public interface IValidationAction
{
    /// <summary>
    /// Executes the action.
    /// </summary>
    /// <param name="isValid">Whether associated validator i
    /// <param name="validationContext">Validation context.
    /// <param name="contextParams">Additional context param
    /// <param name="errors">Validation errors container.
    void Execute(bool isValid, object validationContext, IDi
}
```

12.6. Validator References

Sometimes it is not possible (or desirable) to nest all the validation rules within a single top-level validator group. For example, if you have an object graph where both ObjectA and ObjectB have a reference to ObjectC, you might want to set up validation rules for ObjectC only once and reference them from the validation rules for both ObjectA and ObjectB, instead of duplicating them within both definitions.

The syntax is shown below

```
<v:ref name="referencedValidatorId" context="validationConte</pre>
```

An example is shown below

It is as simple as that — you define validation rules for ObjectC separately and reference them from within other validation groups. Important thing to realize that in most cases you will also want to "narrow" the context for the referenced validator, typically by specifying the name of the property that holds referenced object. In the example above, ObjectA.MyObjectC and ObjectB.ObjectCProperty are both of type ObjectC, which objectC.validator expects to receive as the validation context.

12.7. Progammatic usage

You can also create Validators programmatically using the API. An example is shown below

No matter if you create your validators programmatically or declaratively, you can invoke them in service side code via the 'Validate' method shown above and then handle error conditions. Spring provides AOP parameter validation advice as part of ithe <u>aspect library</u> which may also be useful for performing server-side validation.

12.8. Usage tips within ASP.NET

Now that you know how to configure validation rules, let's see what it takes to evaluate those rules within your typical ASP.NET application and to display error messages.

The first thing you need to do is inject validators you want to use into your ASP.NET page, as shown in the example below:

Once that's done, you need to perform validation in one or more of the page event handlers, which typically looks similar to this:

```
public void SearchForFlights(object sender, EventArgs e)
{
    if (Validate(Controller.Trip, tripValidator))
    {
        Process.SetView(Controller.SearchForFlights());
    }
}
```



Note

Keep in mind that your ASP.NET page needs to extend Spring.Web.UI.Page in order for the code above to work.

Finally, you need to define where validation errors should be displayed by adding one or more <spring:validationError/> and <spring:validationSummary/> controls to the ASP.NET form:

```
<%@ Page Language="c#" MasterPageFile="~/Web/StandardTemplat</pre>
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Cor</pre>
<asp:Content ID="head" ContentPlaceHolderID="head" runat="se</pre>
   <script language="javascript" type="text/javascript">
     <!--
     function showReturnCalendar(isVisible)
         document.getElementById('<%= returningOnDate.Clier
         document.getElementById('returningOnCalendar').sty
     -->
   </script>
</asp:Content>
<asp:Content ID="body" ContentPlaceHolderID="body" runat="se</pre>
   <div style="text-align: center">
       <h4><asp:Label ID="caption" runat="server"></asp:Lab
       <spring:ValidationSummary ID="validationSummary" rur</pre>
         
              <spring:RadioButtonGroup ID="tripMode" r</pre>
                      <asp:RadioButton ID="OneWay" onclick
                      <asp:RadioButton ID="RoundTrip" oncl</pre>
                  </spring:RadioButtonGroup>
              <asp:Label ID="leavingFrom" runat="serve"</pre>
              <asp:DropDownList ID="leavingFromAirport"</pre>
                  <spring:ValidationError id="departureAir</pre>
              <asp:Label ID="goingTo" runat="server"</pre>
              <asp:DropDownList ID="goingToAirportCode"</pre>
```

```
<spring:ValidationError id="destinationA</pre>
             <asp:Label ID="leavingOn" runat="server"</pre>
             <spring:Calendar ID="leavingFromDate" ru</pre>
                <spring:ValidationError id="departureDat"</pre>
             <asp:Label ID="returning0n" runat="serve</pre>
             <div id="returningOnCalendar">
                   <spring:Calendar ID="returningOnDate"</pre>
                   <spring:ValidationError id="returnDa"</pre>
                </div>
             <br/>
                <asp:Button ID="findFlights" runat="serv</pre>
         </div>
   <script language="javascript" type="text/javascript">
        if (document.getElementById('<%= tripMode.ClientID
           showReturnCalendar(false);
        else
           showReturnCalendar(true);
   </script>
</asp:Content>
```

12.8.1. Rendering Validation Errors

Spring.NET allows you to render validation errors within the page in several different ways, and if none of them suits your needs you can implement your own validation errors renderer. Implementations of the

Spring.Web.Validation.IValidationErrorsRenderer that ship with the framework are:

Table 12.3	8. Validation	Renderers
Table 14a	, vanuauvn	1/CHUCLEI 3

Name	Class	
Block	Spring.Web.Validation.DivValidationErrorsRenderer	
Inline	Spring.Web.Validation.SpanValidationErrorsRendere	
Icon	Spring.Web.Validation.IconValidationErrorsRendere	
Icon	Spring.Web.Validation.IconValidationErrorsRe	

These three error renderers should be sufficient for most applications, but in case you want to display errors in some other way you can write your own renderer by implementing Spring.Web.Validation.IValidationErrorsRenderer interface:

```
namespace Spring.Web.Validation
{
    /// <summary>
    /// This interface should be implemented by all validati
    /// </summary>
    /// <remarks>
    /// <para>
    /// Validation errors renderers are used to decouple rer
    /// validation errors controls such as <see cref="Validation"// <see cref="ValidationSummary"/>.
    /// </para>
    /// <para>
    /// <para></para>
```

```
/// This allows users to change how validation errors ar
/// appropriate renderer implementation into the validat
/// Spring.NET dependency injection.
/// </para>
/// </remarks>
public interface IValidationErrorsRenderer
{
    /// <summary>
    /// Renders validation errors using specified <see of
    /// </summary>
    /// <param name="page">Web form instance.</param>
    /// <param name="writer">An HTML writer to use.</param /// <param name="errors">The list of validation error void RenderErrors(Page page, HtmlTextWriter writer, )
}
```

12.8.1.1. Configuring which Error Renderer to use.

The best part of the errors renderer mechanism is that you can easily change it across the application by modifying configuration templates for <spring:validationSummary> and <spring:validationError> controls:

It's as simple as that!

Chapter 13. Aspect Oriented Programming with Spring.NET

13.1. Introduction

Aspect-Oriented Programming (AOP) complements OOP by providing another way of thinking about program structure. Whereas OO decomposes applications into a hierarchy of objects, AOP decomposes programs into aspects or concerns. This enables the modularization of concerns such as transaction management that would otherwise cut across multiple objects (such concerns are often termed *crosscutting* concerns).

One of the key components of Spring.NET is the *AOP framework*. While the Spring.NET IoC container does not depend on AOP, meaning you don't need to use AOP if you don't want to, AOP complements Spring.NET IoC to provide a very capable middleware solution.

AOP is used in Spring.NET:

- To provide declarative enterprise services, especially as a replacement for COM+ declarative services. The most important such service is *declarative transaction management*, which builds on Spring.NET's transaction abstraction. This functionality is planed for an upcoming release of Spring.NET
- To allow users to implement custom aspects, complementing their use of OOP with AOP.

Thus you can view Spring.NET AOP as either an enabling technology that allows Spring.NET to provide declarative transaction management without COM+; or use the full power of the Spring.NET AOP framework to implement custom aspects.

For those who would like to hit the ground running and start exploring how to use Spring's AOP functionality, head on over to Chapter 34, AOP Guide.

13.1.1. AOP concepts

Let us begin by defining some central AOP concepts. These terms are not Spring.NET-specific. Unfortunately, AOP terminology is not particularly intuitive. However, it would be even more confusing if Spring.NET used its own terminology.

• *Aspect*: A modularization of a concern for which the implementation might otherwise cut across multiple objects. Transaction management is a good example of a crosscutting concern in enterprise applications. Aspects are

- implemented using Spring.NET as Advisors or interceptors.
- *Joinpoint*: Point during the execution of a program, such as a method invocation or a particular exception being thrown.
- Advice: Action taken by the AOP framework at a particular joinpoint.
 Different types of advice include "around," "before" and "throws" advice.
 Advice types are discussed below. Many AOP frameworks, including Spring.NET, model an advice as an *interceptor*, maintaining a chain of interceptors "around" the joinpoint.
- *Pointcut*: A set of joinpoints specifying when an advice should fire. An AOP framework must allow developers to specify pointcuts: for example, using regular expressions.
- *Introduction*: Adding methods or fields to an advised class. Spring.NET allows you to introduce new interfaces to any advised object. For example, you could use an introduction to make any object implement an **IAuditable** interface, to simplify the tracking of changes to an object's state.
- *Target object*: Object containing the joinpoint. Also referred to as *advised* or *proxied* object.
- *AOP proxy*: Object created by the AOP framework, including advice. In Spring.NET, an AOP proxy is a dynamic proxy that uses IL code generated at runtime.
- *Weaving*: Assembling aspects to create an advised object. This can be done at compile time (using the Gripper-Loom.NET compiler, for example), or at runtime. Spring.NET performs weaving at runtime.

Different advice types include:

- *Around advice*: Advice that surrounds a joinpoint such as a method invocation. This is the most powerful kind of advice. Around advice will perform custom behaviour before and after the method invocation. They are responsible for choosing whether to proceed to the joinpoint or to shortcut executing by returning their own return value or throwing an exception.
- *Before advice*: Advice that executes before a joinpoint, but which does not have the ability to prevent execution flow proceeding to the joinpoint (unless it throws an exception).

- *Throws advice*: Advice to be executed if a method throws an exception. Spring.NET provides strongly typed throws advice, so you can write code that catches the exception (and subclasses) you're interested in, without needing to cast from Exception.
- *After returning advice*: Advice to be executed after a joinpoint completes normally: for example, if a method returns without throwing an exception.

Spring.NET provides a full range of advice types. We recommend that you use the least powerful advice type that can implement the required behaviour. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you don't need to invoke the <code>proceed()</code> method on the <code>IMethodInvocation</code> used for around advice, and hence can't fail to invoke it.

The pointcut concept is the key to AOP, distinguishing AOP from older technologies offering interception. Pointcuts enable advice to be targeted independently of the OO hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects. Thus pointcuts provide the structural element of AOP.

13.1.2. Spring.NET AOP capabilities

Spring.NET AOP is implemented in pure C#. There is no need for a special compilation process - all weaving is done at runtime. Spring.NET AOP does not need to control or modify the way in which assemblies are loaded, nor does it rely on unmanaged APIs, and is thus suitable for use in any CLR environment.

Spring.NET currently supports interception of method invocations. Field interception is not implemented, although support for field interception could be added without breaking the core Spring.NET AOP APIs.

Field interception arguably violates OO encapsulation. We don't believe it is wise in application development.

Spring.NET provides classes to represent pointcuts and different advice types. Spring.NET uses the term *advisor* for an object representing an aspect, including both an advice and a pointcut targeting it to specific joinpoints.

Different advice types are IMethodInterceptor (from the AOP Alliance interception API); and the advice interfaces defined in Spring.Aop namespace. All advices implement the must AopAlliance. Aop. IAdvice tag interface. Advices supported out the IMethodInterceptor ; IThrowsAdvice: box IBeforeAdvice; and IAfterReturningAdvice. We'll discuss advice types in detail below.

Spring.NET provides a .NET translation of the Java interfaces defined by the <u>AOP Alliance</u>. Around advice must implement the AOP Alliance AopAlliance.Interceptr.IMethodInterceptor interface. Whilst there is wide support for the AOP Alliance in Java, Spring.NET is currently the only .NET AOP framework that makes use of these interfaces. In the short term, this will provide a consistent programming model for those doing development in both .NET and Java, and in the longer term, we hope to see more .NET projects adopt the AOP Alliance interfaces.

The aim of Spring.NET AOP support is not to provide a comprehensive AOP implementation on par with the functionality available in AspectJ. However, Spring.NET AOP provides an excellent solution to most problems in .NET applications that are amenable to AOP.

Thus, it is common to see Spring.NET's AOP functionality used in conjunction with a Spring.NET IoC container. AOP advice is specified using normal object definition syntax (although this allows powerful "autoproxying" capabilities); advice and pointcuts are themselves managed by Spring.NET IoC.

13.1.3. AOP Proxies in Spring.NET

Spring.NET generates AOP proxies at runtime using classes from the System.Reflection.Emit namespace to create necessary IL code for the proxy class. This results in proxies that are very efficient and do not impose any restrictions on the inheritance hierarchy.

Another common approach to AOP proxy implementation in .NET is to use ContextBoundObject and the .NET remoting infrastructure as an interception mechanism. We are not very fond of ContextBoundObject approach because it requires classes that need to be proxied to inherit from the ContextBoundObject either directly or indirectly. In our opinion this an unnecessary restriction that influences how you should design your object model and also excludes applying

AOP to "3rd party" classes that are not under your direct control. Context-bound proxies are also an order of magnitude slower than IL-generated proxies, due to the overhead of the context switching and .NET remoting infrastructure.

Spring.NET AOP proxies are also "smart" - in that because proxy configuration is known during proxy generation, the generated proxy can be optimized to invoke target methods via reflection only when necessary (i.e. when there are advices applied to the target method). In all other cases the target method will be called directly, thus avoiding performance hit caused by the reflective invocation.

Finally, Spring.NET AOP proxies will never return a raw reference to a target object. Whenever a target method returns a raw reference to a target object (i.e. "return this;"), AOP proxy will recognize what happened and will replace the return value with a reference to itself instead.

The current implementation of the AOP proxy generator uses object composition to delegate calls from the proxy to a target object, similar to how you would implement a classic Decorator pattern. This means that classes that need to be proxied have to implement one or more interfaces, which is in our opinion not only a less-intruding requirement than ContextBoundObject inheritance requirements, but also a good practice that should be followed anyway for the service classes that are most common targets for AOP proxies.

In a future release we will implement proxies using inheritance, which will allow you to proxy classes without interfaces as well and will remove some of the remaining raw reference issues that cannot be solved using composition-based proxies.

13.2. Pointcut API in Spring.NET

Let's look at how Spring.NET handles the crucial pointcut concept.

13.2.1. Concepts

Spring.NET's pointcut model enables pointcut reuse independent of advice types. It's possible to target different advice using the same pointcut.

The **Spring.Aop.IPointcut** interface is the central interface, used to target advices to particular types and methods. The complete interface is shown below:

```
public interface IPointcut
{
    ITypeFilter TypeFilter { get; }

    IMethodMatcher MethodMatcher { get; }
}
```

Splitting the **IPointcut** interface into two parts allows reuse of type and method matching parts, and fine-grained composition operations (such as performing a "union" with another method matcher).

The ITypeFilter interface is used to restrict the pointcut to a given set of target classes. If the Matches() method always returns true, all target types will be matched:

```
public interface ITypeFilter
{
    bool Matches(Type type);
}
```

The **IMethodMatcher** interface is normally more important. The complete interface is shown below:

```
public interface IMethodMatcher
{
   bool IsRuntime { get; }
   bool Matches(MethodInfo method, Type targetType);
```

```
bool Matches(MethodInfo method, Type targetType, object[
}
```

The Matches (MethodInfo, Type) method is used to test whether this pointcut will ever match a given method on a target type. This evaluation can be performed when an AOP proxy is created, to avoid the need for a test on every method invocation. If the 2-argument matches method returns true for a given method, and the IsRuntime property for the IMethodMatcher returns true, the 3-argument matches method will be invoked on every method invocation. This enables a pointcut to look at the arguments passed to the method invocation immediately before the target advice is to execute.

Most IMethodMatchers are static, meaning that their IsRuntime property returns false. In this case, the 3-argument Matches method will never be invoked.

Whenever possible, try to make pointcuts static... this allows the AOP framework to cache the results of pointcut evaluation when an AOP proxy is created.

13.2.2. Operations on pointcuts

Spring.NET supports operations on pointcuts: notably, *union* and *intersection*.

Union means the methods that either pointcut matches.

Intersection means the methods that both pointcuts match.

Union is usually more useful.

Pointcuts can be composed using the static methods in the *Spring.Aop.Support.Pointcuts* class, or using the *ComposablePointcut* class in the same namespace.

13.2.3. Convenience pointcut implementations

Spring.NET provides several convenient pointcut implementations. Some can be used out of the box; others are intended to be subclassed in application-specific pointcuts.

13.2.3.1. Static pointcuts

Static pointcuts are based on method and target class, and cannot take into account the method's arguments. Static pointcuts are sufficient--and best--for

most usages. It's possible for Spring.NET to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

Let's consider some static pointcut implementations included with Spring.NET.

13.2.3.1.1. Regular expression pointcuts

One obvious way to specify static pointcuts is using regular expressions. Several AOP frameworks besides Spring.NET make this possible. The Spring.Aop.Support.SdkRegularExpressionMethodPoclass is a generic regular expression pointcut, that uses the regular expression classes from the .NET BCL.

Using this class, you can provide a list of pattern Strings. If any of these is a match, the pointcut will evaluate to true (so the result is effectively the union of these pointcuts.). The matching is done against the full class name so you can use this pointcut if you would like to apply advice to all the classes in a particular namespace.

The usage is shown below:

As a convenience, Spring provides the RegularExpressionMethodPointcutAdvisor class that allows us to reference an IAdvice instance as well as defining the pointcut rules (remember that an IAdvice instance can be an interceptor, before advice, throws advice etc.) This simplifies wiring, as the one object serves as both pointcut and advisor, as shown below:

The RegularExpressionMethodPointcutAdvisor class can be used with any Advice type.

If you only have one pattern you can use the property name pattern and specify a single value instead of using the property name patterns and specifying a list.

You also Regex object from may specify the a System. Text. Regular Expressions namespace. The built in RegexConverter class will perform the conversion. See Section 6.4, "Built-in TypeConverters" for more information on Spring's build in type converters. The Regex object is created as any other object within the IoC container. Using an inner-object definition for the Regex object is a handy way to keep the definition PointcutAdvisor declaration. Note that the close the class SdkRegularExpressionMethodPointcut has DefaultOptions property to set the regular expression options if they are not explicitly specified in the constructor.

13.2.3.1.2. Attribute pointcuts

Pointcuts can be specified by matching an attribute type that is associated with a method. Advice associated with this pointcut can then read the metadata associated with the attribute to configure itself. The class AttributeMatchMethodPointcut provides this functionality.

Sample usage that will match all methods that have the attribute Spring.Attributes.CacheAttribute is shown below.

This can be used with a **DefaultPointcutAdvisor** as shown below

where aspNetCacheAdvice is an implementation of an IMethodInterceptor that caches method return values. See the SDK docs for Spring.Aop.Advice.CacheAdvice for more information on this particular advice.

As a convenience the class AttributeMatchMethodPointcutAdvisor is provided to defining an attribute based Advisor as a somewhat shorter alternative to using the generic DefaultPointcutAdvisor. An example is shown below.

13.2.3.2. Dynamic Pointcuts

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method *arguments*, as well as static information. This means that they

must be evaluated with every method invocation; the result cannot be cached, as arguments will vary.

The main example is the **CONTROL** flow pointcut.

13.2.3.2.1. Control Flow Pointcuts

Spring.NET control flow pointcuts are conceptually similar to AspectJ cflow pointcuts, although less powerful. (There is currently no way to specify that a pointcut executes below another pointcut.). A control flow pointcut is dynamic because it is evaluated against the current call stack for each method invocation. For example, if method ClassA.A() calls ClassB.B() then the execution of ClassB.B() has occurred in ClassA.A()'s control flow. A control flow pointcut allows advice to be applied to the method ClassA.A() but only when called from ClassB.B() and not when ClassA.A() is executed from another call stack. specified Control flow pointcuts the are Spring.Aop.Support.ControlFlowPointcut class.



Control flow pointcuts are significantly more expensive to evaluate at runtime than even other dynamic pointcuts.

When using control flow point cuts some attention should be paid to the fact that at runtime the JIT compiler can inline the methods, typically for increased performance, but with the consequence that the method no longer appears in the current call stack. This is because inlining takes the callee's IL code and inserts it into the caller's IL code effectively removing the method call. The information returned from System. Diagnostics. StackTrace, used in the implementation of ControlFlowPointcut is subject to these optimizations and therefore a control flow pointcut will not match if the method has been inlined.

Generally speaking, a method will be a candidate for inlining when its code is 'small', just a few lines of code (less than 32 bytes of IL). For some interesting reading on this process read David Notario's blog entries (JIT Optimizations I and JIT Optimizations II). Additionally, when an assembly is compiled with a Release configuration the assembly metadata instructs the CLR to enable JIT

optimizations. When compiled with a Debug configuration the CLR will disable (some?) these optimizations. Empirically, method inlining is turned off in a Debug configuration.

The way to ensure that your control flow pointcut will not be overlooked because of method inlining is to apply the System.Runtime.CompilerServices.MethodImplAttri attribute with the value MethodImplOptions.NoInlining. In this (somewhat artificial) simple example, if the code is compiled in release mode it will not match a control flow pointcut for the method "GetAge".

```
public int GetAge(IPerson person)
{
    return person.GetAge();
}
```

However, applying the attributes as shown below will prevent the method from being inlined even in a release build.

```
[MethodImpl(MethodImplOptions.NoInlining)]
public int GetAge(IPerson person)
{
    return person.GetAge();
}
```

13.2.4. Custom pointcuts

Because pointcuts in Spring.NET are .NET types, rather than language features (as in AspectJ) it is possible to declare custom pointcuts, whether static or dynamic. However, there is no support out of the box for the sophisticated pointcut expressions that can be coded in the AspectJ syntax. However, custom pointcuts in Spring.NET can be as arbitrarily complex as any object model.

Spring.NET provides useful pointcut superclasses to help you to implement your own pointcuts.

Because static pointcuts are the most common and generally useful pointcut type, you'll probably subclass **StaticMethodMatcherPointcut**, as shown below. This requires you to implement just one abstract method (although it is possible to override other methods to customize behaviour):

13.3. Advice API in Spring.NET

Let's now look at how Spring.NET AOP handles advice.

13.3.1. Advice Lifecycle

Spring.NET advices can be shared across all advised objects, or unique to each advised object. This corresponds to *per-class* or *per-instance* advice.

Per-class advice is used most often. It is appropriate for generic advice such as transaction advisors. These do not depend on the state of the proxied object or add new state; they merely act on the method and arguments.

Per-instance advice is appropriate for introductions, to support mixins. In this case, the advice adds state to the proxied object.

It's possible to use a mix of shared and per-instance advice in the same AOP proxy.

13.3.2. Advice types

Spring.NET provides several advice types out of the box, and is extensible to support arbitrary advice types. Let us look at the basic concepts and standard advice types.

13.3.2.1. Interception Around Advice

The most fundamental advice type in Spring.NET is interception around advice.

Spring.NET is compliant with the AOP Alliance interface for around advice using method interception. Around advice is implemented using the following interface:

```
public interface IMethodInterceptor : IInterceptor
{
   object Invoke(IMethodInvocation invocation);
}
```

The IMethodInvocation argument to the Invoke() method exposes the method being invoked; the target joinpoint; the AOP proxy; and the arguments to the method. The Invoke() method should return the invocation's result: the return value of the joinpoint.

A simple IMethodInterceptor implementation looks as follows:

```
public class DebugInterceptor : IMethodInterceptor {
    public object Invoke(IMethodInvocation invocation) {
        Console.WriteLine("Before: invocation=[{0}]", invocation=[total]", invocati
```

Note the call to the IMethodInvocation's Proceed() method. This proceeds down the interceptor chain towards the joinpoint. Most interceptors will invoke this method, and return its return value. However, an IMethodInterceptor, like any around advice, can return a different value or throw an exception rather than invoke the Proceed() method. However, you don't want to do this without good reason!

13.3.2.2. Before advice

A simpler advice type is a **before advice**. This does not need an **IMethodInvocation** object, since it will only be called before entering the method.

The main advantage of a before advice is that there is no need to invoke the Proceed() method, and therefore no possibility of inadvertently failing to proceed down the interceptor chain.

The IMethodBeforeAdvice interface is shown below.

```
public interface IMethodBeforeAdvice : IBeforeAdvice
{
   void Before(MethodInfo method, object[] args, object tar
}
```

Note the return type is **VOid**. Before advice can insert custom behaviour before the joinpoint executes, but cannot change the return value. If a before advice throws an exception, this will abort further execution of the interceptor chain. The exception will propagate back up the interceptor chain. If it is unchecked, or

on the signature of the invoked method, it will be passed directly to the client; otherwise it will be wrapped in an unchecked exception by the AOP proxy.

An example of a before advice in Spring.NET, which counts all methods that return normally:

Before advice can be used with any pointcut.

13.3.2.3. Throws advice

Throws advice is invoked after the return of the joinpoint if the joinpoint threw an exception. The Spring.Aop.IThrowsAdvice interface does not contain any methods: it is a tag interface identifying that the implementing advice object implements one or more typed throws advice methods. These throws advice methods must be of the form:

```
AfterThrowing([MethodInfo method, Object[] args, Object targ
```

Throws-advice methods must be named 'AfterThrowing'. The return value will be ignored by the Spring.NET AOP framework, so it is typically VOid. With regard to the method arguments, only the last argument is required. Thus there are *exactly* one *or* four arguments, depending on whether the advice method is interested in the method, method arguments and the target object.

The following method snippets show examples of throws advice.

This advice will be invoked if a **RemotingException** is thrown (including subclasses):

```
public class RemoteThrowsAdvice : IThrowsAdvice {
    public void AfterThrowing(RemotingException ex) {
        // Do something with remoting exception
    }
}
```

The following advice is invoked if a Sqlexception is thrown. Unlike the above advice, it declares 4 arguments, so that it has access to the invoked method, method arguments and target object:

The final example illustrates how these two methods could be used in a single class, which handles both RemotingException and Sqlexception. Any number of throws advice methods can be combined in a single class, as can be seen in the following example.

```
public class CombinedThrowsAdvice : IThrowsAdvice {
    public void AfterThrowing(RemotingException ex) {
        // Do something with remoting exception
    }
    public void AfterThrowing(MethodInfo method, object[] ar
        // Do something will all arguments
    }
}
```

Finally, it is worth stating that throws advice is only applied to the actual exception being thrown. What does this mean? Well, it means that if you have defined some throws advice that handles RemotingExceptions, the applicable AfterThrowing method will **only** be invoked if the type of the thrown exception is RemotingException... if a

RemotingException has been thrown and subsequently wrapped inside another exception before the exception bubbles up to the throws advice interceptor, then the throws advice that handles RemotingExceptions will **never** be called. Consider a business method that is advised by throws advice that handles RemotingExceptions; if during the course of a method invocation said business method throws a RemoteException... and subsequently wraps said RemotingException inside a business-specific BadConnectionException (see the code snippet below) before throwing the exception, then the throws advice will never be able to respond to the RemotingException... because all the throws advice sees is a BadConnectionException. The fact that the RemotingException inside the wrapped up BadConnectionException is immaterial.

```
public void BusinessMethod()
{
    try
    {
        // do some business operation...
    }
    catch (RemotingException ex)
    {
        throw new BadConnectionException("Couldn't connectionException("Couldn't connectionException("Couldn't connectionException("Couldn't connectionException("Couldn't connectionException("Couldn't connectionException("Couldn't connectionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExceptionExce
```



Note

Please note that throws advice can be used with any pointcut.

13.3.2.4. After Returning advice

An after returning advice in Spring.NET must implement the Spring.Aop.IAfterReturningAdvice interface, shown below:

```
public interface IAfterReturningAdvice : IAdvice
{
  void AfterReturning(object returnValue, MethodBase method,
```

}

An after returning advice has access to the return value (which it cannot modify), invoked method, methods arguments and target.

The following after returning advice counts all successful method invocations that have not thrown exceptions:

This advice doesn't change the execution path. If it throws an exception, this will be thrown up the interceptor chain instead of the return value.



Note

Please note that after-returning advice can be used with any pointcut.

13.3.2.5. Advice Ordering

When multiple pieces of advice want to run on the same joinpoint the precedence is determined by having the advice implement the IOrdered interface or by specifying order information on an advisor.

13.3.2.6. Introduction advice

Spring.NET allows you to add new methods and properties to an advised class. This would typically be done when the functionality you wish to add is a crosscutting concern and want to introduce this functionality as a change to the static structure of the class hierarchy. For example, you may want to cast objects to the introduction interface in your code. Introductions are also a means to emulate multiple inheritance.

Introduction advice is defined by using a normal interface declaration that implements the tag interface **IAdvice**.



Note

The need for implementing this marker interface will likely be removed in future versions.

As an example, consider the interface **IAuditable** that describes the last modified time of an object.

```
public interface IAuditable : IAdvice
{
    DateTime LastModifiedDate
    {
        get;
        set;
    }
}
```

where

```
public interface IAdvice
{
}
```

Access to the advised object can be obtained by implementing the interface ITargetAware

```
public interface ITargetAware
{
   IAopProxy TargetProxy
   {
      set;
   }
}
```

with the **IAOPProxy** reference providing a layer of indirection through which the advised object can be accessed.

```
public interface IAopProxy
{
   object GetProxy();
}
```

A simple class that demonstrates this functionality is shown below.

```
public interface IAuditable : IAdvice, ITargetAware
{
    DateTime LastModifiedDate
    {
        get;
        set;
    }
}
```

A class that implements this interface is shown below.

```
public class AuditableMixin : IAuditable
{
    private DateTime date;
    private IAopProxy targetProxy;

    public AuditableMixin()
    {
        date = new DateTime();
    }

    public DateTime LastModifiedDate
    {
        get { return date; }
        set { date = value; }
    }

    public IAopProxy TargetProxy
    {
        set { targetProxy = value; }
}
```

Introduction advice is not associated with a pointcut, since it applies at the class

and not the method level. As such, introductions use their own subclass of the interface <code>IAdvisor</code>, namely <code>IIntroductionAdvisor</code>, to specify the types that the introduction can be applied to.

```
public interface IIntroductionAdvisor : IAdvisor
{
    ITypeFilter TypeFilter { get; }

    Type[] Interfaces { get; }

    void ValidateInterfaces();
}
```

The **TypeFilter** property returns the filter that determines which target classes this introduction should apply to.

The **Interfaces** property returns the interfaces introduced by this advisor.

The ValidateInterfaces() method is used internally to see if the introduced interfaces can be implemented by the introduction advice.

Spring.NET provides a default implementation of this interface (the <code>DefaultIntroductionAdvisor</code> class) that should be sufficient for the majority of situations when you need to use introductions. The most simple implementation of an introduction advisor is a subclass that simply passes a new instance the base constructor. Passing a new instance is important since we want a new instance of the mixin classed used for each advised object.

```
public class AuditableAdvisor : DefaultIntroductionAdvisor
{
   public AuditableAdvisor() : base(new AuditableMixin())
   {
    }
}
```

Other constructors let you explicitly specify the interfaces of the class that will be introduced. See the SDK documentation for more details.

We can apply this advisor Programatically, using the IAdvised.AddIntroduction(), method, or (the recommended way) in XML configuration using the IntroductionNames property on

ProxyFactoryObject, which will be discussed later.

Unlike the AOP implementation in the Spring Framework for Java, introduction advice in Spring.NET is not implemented as a specialized type of interception advice. The advantage of this approach is that introductions are not kept in the interceptor chain, which allows some significant performance optimizations. When a method is called that has no interceptors, a direct call is used instead of reflection regardless of whether the target method is on the target object itself or one of the introductions. This means that introduced methods perform the same as target object methods, which could be useful for adding introductions to fine grained objects. The disadvantage is that if the mixin functionality would benefit from having access to the calling stack, it is not available. Introductions with this functionality will be addressed in a future version of Spring.NET AOP.

13.4. Advisor API in Spring.NET

In Spring.NET, an advisor is a modularization of an aspect. Advisors typically incorporate both an advice and a pointcut.

Apart from the special case of introductions, any advisor can be used with any advice.

Spring.Aop.Support.DefaultPointcutAdvisor class is the most commonly used advisor implementation. For example, it can be used with a IMethodInterceptor, IBeforeAdvice or IThrowsAdvice and any pointcut definition.

Other convenience implementations provided are: AttributeMatchMethodPointcutAdvisor shown in usage previously in <u>Section 13.2.3.1.2</u>, "Attribute pointcuts" for use with attribute based pointcuts. RegularExpressionMethodPointcutAdvisor that will apply pointcuts based on the matching a regular expression to method names.

It is possible to mix advisor and advice types in Spring.NET in the same AOP proxy. For example, you could use a interception around advice, throws advice and before advice in one proxy configuration: Spring.NET will automatically create the necessary interceptor chain.

13.5. Using the ProxyFactoryObject to create AOP proxies

If you're using the Spring.NET IoC container for your business objects - generally a good idea - you will want to use one of Spring.NET's AOP-specific **IFactoryObject** implementations (remember that a factory object introduces a layer of indirection, enabling it to create objects of a different type - Section 5.3.9, "Setting a reference using the members of other objects and classes.").

The basic way to create an AOP proxy in Spring.NET is to use the Spring.Aop.Framework.ProxyFactoryObject class. This gives complete control over ordering and application of the pointcuts and advice that will apply to your business objects. However, there are simpler options that are preferable if you don't need such control.

13.5.1. Basics

The ProxyFactoryObject, like other Spring.NET IFactoryObject implementations, introduces a level of indirection. If you define a ProxyFactoryObject with name foo, what objects referencing foo see is not the ProxyFactoryObject instance itself, but an object created by the ProxyFactoryObject's implementation of the GetObject() method. This method will create an AOP proxy wrapping a target object.

One of the most important benefits of using a ProxyFactoryObject or other IoC-aware classes that create AOP proxies, is that it means that advice and pointcuts can also be managed by IoC. This is a powerful feature, enabling certain approaches that are hard to achieve with other AOP frameworks. For example, an advice may itself reference application objects (besides the target, which should be available in any AOP framework), benefiting from all the pluggability provided by Dependency Injection.

13.5.2. ProxyFactoryObject Properties

Like most IFactoryObject implementations provided with Spring.NET, the ProxyFactoryObject is itself a Spring.NET configurable object. Its properties are used to:

• Specify the target object that is to be proxied.

• Specify the advice that is to be applied to the proxy.

Some key properties are inherited from the Spring.Aop.Framework.ProxyConfig class: this class is the superclass for all AOP proxy factories in Spring.NET. Some of the key properties include:

- ProxyTargetType: a boolean value that should be set to true if the target class is to be proxied directly, as opposed to just proxying the interfaces exposed on the target class.
- Optimize: whether to apply aggressive optimization to created proxies. Don't use this setting unless you understand how the relevant AOP proxy handles optimization. The exact meaning of this flag will differ between proxy implementations and will generally result in a trade off between proxy creation time and runtime performance. Optimizations may be ignored by certain proxy implementations and may be disabled silently based on the value of other properties such as ExposeProxy.
- **ISFrozen**: whether advice changes should be disallowed once the proxy factory has been configured. The default is false.
- ExposeProxy: whether the current proxy should be exposed via the **AopContext** so that it can be accessed by the target. (It's available via **IMethodInvocation** the without the need for the AopContext.) If a target needs obtain the to and proxy ExposeProxy true. is the the target can use AopContext.CurrentProxy property.
- AopProxyFactory: the implementation of IAopProxyFactory to use when generating a proxy. Offers a way of customizing whether to use remoting proxies, IL generation or any other proxy strategy. The default implementation will use IL generation to create composition-based proxies.

Other properties specific to the ProxyFactoryObject class include:

- ProxyInterfaces: the array of string interface names we're proxying.
- InterceptorNames: string array of IAdvisor, interceptor

or other advice names to apply. Ordering is significant... first come, first served that is. The first interceptor in the list will be the first to be able to interceptor the invocation (assuming it concerns a regular MethodInterceptor or BeforeAdvice).

The names are object names in the current container, including objectnames from container hierarchies. You can't mention object references here since doing so would result in the ProxyFactoryObject ignoring the singleton setting of the advise.

- IntroductionNames: The names of objects in the container that will be used as introductions to the target object. If the object referred to by name does not implement the IIntroductionAdvisor it will be the default of passed to constructor DefaultIntroductionAdvisor and all of objects interfaces will be added to the target object. Objects that implement the IIntroductionAdvisor interface will be used as is, giving you a finer level of control over what interfaces you may want to expose and the types for which they will be matched against.
- IsSingleton: whether or not the factory should return a single proxy object, no matter how often the GetObject() method is called. Several IFactoryObject implementations offer such a method. The default value is true. If you would like to be able to apply advice on a per-proxy object basis, use a IsSingleton value of false and a IsFrozen value of false. If you want to use stateful advice--for example, for stateful mixins--use prototype advices along with a IsSingleton value of false.

13.5.3. Proxying Interfaces

Let's look at a simple example of ProxyFactoryObject in action. This example involves:

- A target object that will be proxied. This is the "personTarget" object definition in the example below.
- An IAdvisor and an IInterceptor used to provide advice.
- An AOP proxy object definition specifying the target object (the

personTarget object) and the interfaces to proxy, along with the advices to apply.

```
<object id="personTarget" type="MyCompany.MyApp.Person, MyCo"</pre>
    property name="name" value="Tony"/>
    cproperty name="age" value="51"/>
</object>
<object id="myCustomInterceptor" type="MyCompany.MyApp.MyCus</pre>
    cproperty name="customProperty" value="configuration str
</object>
<object id="debugInterceptor" type="Spring.Aop.Advice.DebugA</pre>
</object>
<object id="person" type="Spring.Aop.Framework.ProxyFactory()</pre>
    roperty name="proxyInterfaces" value="MyCompany.MyApp.
    property name="target" ref="personTarget"/>
    property name="interceptorNames">
        st>
            <value>debugInterceptor</value>
            <value>myCustomInterceptor</value>
        </list>
    </property>
</object>
```

Note that the InterceptorNames property takes a list of Strings: the object names of the interceptor or advisors in the current context. Advisors, interceptors, before, after returning and throws advice objects can be used. The ordering of advisors is significant.

You might be wondering why the list doesn't hold object references. The reason for this is that if the ProxyFactoryObject's singleton property is set to false, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, an independent instance would need to be returned, so it's necessary to be able to obtain an instance of the prototype from the context; holding a reference isn't sufficient.

The "person" object definition above can be used in place of an **IPerson** implementation, as follows:

```
IPerson person = (IPerson) factory.GetObject("person");
```

Other objects in the same IoC context can express a strongly typed dependency on it, as with an ordinary .NET object:

The PersonUser class in this example would expose a property of type IPerson. As far as it's concerned, the AOP proxy can be used transparently in place of a "real" person implementation. However, its type would be a proxy type. It would be possible to cast it to the IAdvised interface (discussed below).

It's possible to conceal the distinction between target and proxy using an anonymous *inline object*, as follows. (for more information on inline objects see <u>Section 5.3.2.3, "Inline objects"</u>.) Only the ProxyFactoryObject definition is different; the advice is included only for completeness:

This has the advantage that there's only one object of type Person: useful if we want to prevent users of the application context obtaining a reference to the un-advised object, or need to avoid any ambiguity with Spring IoC *autowiring*. There's also arguably an advantage in that the ProxyFactoryObject definition is self-contained. However, there are times when being able to obtain the unadvised target from the factory might actually be an *advantage*: for example, in certain test scenarios.

13.5.1. Applying advice on a per-proxy basis.

Let's look at an example of configuring the proxy objects retrieved from ProxyFactoryObject.

```
<!-- create the object to reference -->
<object id="RealObjectTarget" type="MyRealObject"
<!-- create the proxied object for everyone to use
<object id="MyObject" type="Spring.Aop.Framework.F
<pre>cproperty name="proxyInterfaces" value="MyInterfaces" value="false"/>
cproperty name="isSingleton" value="false"/>
cproperty name="targetName" value="RealObjectTargees"
```

If you are using a prototype as the target you must set the TargetName property with the name/object id of your object and not use the property Target with a reference to that object. This will then allow a new proxy to be created around a new prototype target instance.

Consider the above Spring.Net object configuration. Notice that the

IsSingleton property of the ProxyFactoryObject instance is set to false. This means that each proxy object will be unique. Thus, you can configure each proxy object with its' own individual advice(s) using the following syntax

```
// Will return un-advised instance of proxy object
MyInterface myProxyObject1 = (MyInterface)ctx.GetObject("MyC
// myProxyObject1 instance now has an advice attached to it.
IAdvised advised = (IAdvised)myProxyObject1;
advised.AddAdvice( new DebugAdvice() );
// Will return a new, un-advised instance of proxy object
MyInterface myProxyObject2 = (MyInterface)ctx.GetObject("MyComparison of the comparison of the comparison
```

13.5.4. Proxying Classes

What if you need to proxy a class, rather than one or more interfaces?

Imagine that in our example above, there was no IPerson interface, rather we needed to advise a class called Person that didn't implement any business interface. In this case the ProxyFactoryObject will proxy all public virtual methods and properties if no interfaces are explicitly specified or if no interfaces are found to be present on the target object. One can configure Spring.NET to force the use of class proxies, rather than interface proxies, by setting the ProxyTargetType property on the ProxyFactoryObject above to true.

Class proxying works by generating a subclass of the target class at runtime. Spring.NET configures this generated subclass to delegate method calls to the original target: the subclass is used to implement the *Decorator* pattern, weaving in the advice.

Class proxying should generally be transparent to users. However, there is an important issue to consider: *Non-Virtual methods can't be advised*, *as they can't be overridden*. This may be a limiting factor when using existing code as it has been common practice not to declare methods as virtual by default.

13.5.5. Concise proxy definitions

Especially when defining transactional proxies, if you do not make use of the transaction namespace, you may end up with many similar proxy definitions. The use of parent and child object definitions, along with inner object definitions, can result in much cleaner and more concise proxy definitions.

First a parent, template, object definition is created for the proxy:

This will never be instantiated itself, so may actually be incomplete. Then each proxy which needs to be created is just a child object definition, which wraps the target of the proxy as an inner object definition, since the target will never be used on its own anyway.

It is of course possible to override properties from the parent template, such as in this case, the transaction propagation settings:

Note that in the example above, we have explicitly marked the parent object definition as abstract by using the abstract attribute, as described previously, so that it may not actually ever be instantiated. Application contexts (but not simple object factories) will by default pre-instantiate all singletons. It is therefore important (at least for singleton object) that if you have a (parent) object definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the abstract attribute to true, otherwise the application context will actually try to pre-instantiate it.

13.6. Proxying mechanisms

Spring creates AOP proxies built at runtime through the use of the TypeBuilder API.

Two types of proxies can be created, composition based or inheritance based. If the target object implements at least one interface then a composition based proxy will be created, otherwise an inheritance based proxy will be created.

The composition based proxy is implemented by creating a type that implements all the interfaces specified on the target object. The actual class name of this dynamic type is 'GUID' like. A private field holds the target object and the dynamic type implementation will first execute any advice before or after making the target object method call on the target object.

The inheritance based mechanism creates a dynamic type where that inherits from the target type. This lets you downcast to the target type if needed. Please note that in both cases a target method implementation that calls other methods on the target object will not be advised. To force inheritance based proxies you should either set the ProxyTargetType to true property of a ProxyFactory or set the XML namespace element proxy-target-type = true when using an AOP schema based configuration.



An important alternative approach to inheritance based proxies is disucssed in the next section.

In .NET 2.0 you can define the assembly level attribute, Internals Visible To, to allow access of internal interfaces/classes to specified 'friend' assemblies. If you need to create an AOP proxy on an internal class/interface add the following InternalsVisibleTo("Spring.Proxy")] code. [assembly: and InternalsVisibleTo("Spring.DynamicReflection")] to your to AssemblyInfo file.

13.6.1. InheritanceBasedAopConfigurer

There is an important limitation in the inheritance based proxy as described above, all methods that manipulate the state of the object should be declared as virtual. Otherwise some method invocations get directed to the private 'target' field member and others to the base class. Winform object are an example of case where this approach does not apply. To address this limitation, a new postprocessing mechanism was introduced in version 1.2 that creates a proxy type without the private 'target' field. Interception advice is added directly in the method body before invoking the base class method.

To use this new inheritance based proxy described in the note above, declare an instance of the InheritanceBasedAopConfigurer, and IObjectFactoryPostProcessor, in yoru configuration file. Here is an example.

This configuration style is similar to the autoproxy by name approach described here and is particuarly appropriate when you want to apply advice to WinForm classes.

13.7. Creating AOP Proxies Programatically with the ProxyFactory

It's easy to create AOP proxies Programatically using Spring.NET. This enables you to use Spring.NET AOP without dependency on Spring.NET IoC.

The following listing shows creation of a proxy for a target object, with one interceptor and one advisor. The interfaces implemented by the target object will automatically be proxied:

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceI
factory.AddAdvice(myMethodInterceptor);
factory.AddAdvisor(myAdvisor);
IBusinessInterface tb = (IBusinessInterface) factory.GetProx
```

The first step is to construct an object of type Spring.Aop.Framework.ProxyFactory. You can create this with a target object, as in the above example, or specify the interfaces to be proxied in an alternate constructor.

You can add interceptors or advisors, and manipulate them for the life of the ProxyFactory.

There are also convenience methods on ProxyFactory (inherited from AdvisedSupport) allowing you to add other advice types such as before and throws advice. AdvisedSupport is the superclass of both ProxyFactory and ProxyFactoryObject.



Note

Integrating AOP proxy creation with the IoC framework is best practice in most applications. We recommend that you externalize configuration from .NET code with AOP, as in general.

13.8. Manipulating Advised Objects

However you create AOP proxies, you can manipulate them using the Spring.Aop.Framework.IAdvised interface. Any AOP proxy can be cast to this interface, whatever other interfaces it implements. This interface includes the following methods and properties:

```
public interface IAdvised
{
    IAdvisor[] Advisors {
                                get; }
    IIntroductionAdvisor[] Introductions { get; }
         AddInterceptor(IInterceptor interceptor);
    void
         AddInterceptor(int pos, IInterceptor interceptor);
    void
         AddAdvisor(IAdvisor advisor);
    void
         AddAdvisor(int pos, IAdvisor advisor);
    void
         AddIntroduction(IIntroductionAdvisor advisor);
         AddIntroduction(int pos, IIntroductionAdvisor advi
    void
    int IndexOf(IAdvisor advisor);
    int IndexOf(IIntroductionAdvisor advisor);
    bool RemoveAdvisor(IAdvisor advisor);
    void RemoveAdvisor(int index);
    bool RemoveInterceptor(IInterceptor interceptor);
    bool RemoveIntroduction(IIntroductionAdvisor advisor);
    void RemoveIntroduction(int index);
    void ReplaceIntroduction(int index, IIntroductionAdvisor
```

```
bool ReplaceAdvisor(IAdvisor a, IAdvisor b);
}
```

The Advisors property will return an IAdvisor for every advisor, interceptor or other advice type that has been added to the factory. If you added an IAdvisor, the returned advisor at this index will be the object that you added. If you added an interceptor or other advice type, Spring.NET will have wrapped this in an advisor with a IPointcut that always returns true. Thus if you added an IMethodInterceptor, the advisor returned for this index will be a DefaultPointcutAdvisor returning your IMethodInterceptor and an IPointcut that matches all types and methods.

The AddAdvisor() methods can be used to add any IAdvisor. Usually this will be the generic DefaultPointcutAdvisor, which can be used with any advice or pointcut (but not for introduction).

By default, it's possible to add or remove advisors or interceptors even once a proxy has been created. The only restriction is that it's impossible to add or remove an introduction advisor, as existing proxies from the factory will not show the interface change. (You can obtain a new proxy from the factory to avoid this problem.)

It's questionable whether it's advisable (no pun intended) to modify advice on a business object in production, although there are no doubt legitimate usage cases. However, it can be very useful in development: for example, in tests. I have sometimes found it very useful to be able to add test code in the form of an interceptor or other advice, getting inside a method invocation I want to test. (For example, the advice can get inside a transaction created for that method: for example, to run SQL to check that a database was correctly updated, before marking the transaction for roll back.)

Depending on how you created the proxy, you can usually set a Frozen flag, in which case the IAdvised Isfrozen property will return true, and any attempts to modify advice through addition or removal will result in an AopConfigException. The ability to freeze the state of an advised object is useful in some cases: For example, to prevent calling code removing a security interceptor.

13.9. Using the "autoproxy" facility

So far we've considered explicit creation of AOP proxies using a ProxyFactoryObject or similar factory objects. For applications that would like create many AOP proxies, say across all the classes in a service layer, this approach can lead to a lengthy configuration file. To simplify the creation of many AOP proxies Spring provides "autoproxy" capabilities that will automatically proxy object definitions based on higher level criteria that will group together multiple objects as candidates to be proxied.

This functionality is built on Spring "object post-processor" infrastructure, which enables modification of any object definition as the container loads. Refer to Section 5.9.1, "Customizing objects with IObjectPostProcessors" for general information on object post-processors.

In this model, you set up some special object definitions in your XML object definition file configuring the auto proxy infrastructure. This allows you just to declare the targets eligible for autoproxying: you don't need to use ProxyFactoryObject.

- Using an autoproxy creator that refers to specific objects in the current context.
- A special case of autoproxy creation that deserves to be considered separately; autoproxy creation driven by source-level attributes.

Autoproxying in general has the advantage of making it impossible for callers or dependencies to obtain an un-advised object. Calling GetObject("MyBusinessObject1") on an ApplicationContext will return an AOP proxy, not the target business object. The "inline object" idiom shown earlier in Section 13.5.3, "Proxying Interfaces" also offers this benefit.)

13.9.1. Autoproxy object definitions

The namespace Spring.Aop.Framework.AutoProxy provides generic autoproxy infrastructure, should you choose to write your own autoproxy implementations, as well as several out-of-the-box implementations. Two implementations are provided, ObjectNameAutoProxyCreator and DefaultAdvisorAutoProxyCreator. These are discussed in the following sections.

13.9.1.1. ObjectNameAutoProxyCreator

The **ObjectNameAutoProxyCreator** automatically creates AOP proxies for object with names matching literal values or wildcards. The pattern matching expressions supported are of the form "*name", "name*", and "*name*" and exact name matching, i.e. "name". The following simple classes are used to demonstrate this autoproxy functionality.

```
public enum Language
{
    English = 1,
    Portuguese = 2,
    Italian = 3
}
public interface IHelloWorldSpeaker
{
   void SayHello();
}
public class HelloWorldSpeaker : IHelloWorldSpeaker
{
    private Language language;
    public Language Language
        set { language = value; }
        get { return language; }
    }
    public void SayHello()
        switch (language)
            case Language. English:
                Console.WriteLine("Hello World!");
                break;
            case Language.Portuguese:
                Console.WriteLine("Oi Mundo!");
```

```
break;
            case Language.Italian:
                Console.WriteLine("Ciao Mondo!");
                break;
        }
    }
}
public class DebugInterceptor : IMethodInterceptor
{
    public object Invoke(IMethodInvocation invocation)
    {
        Console.WriteLine("Before: " + invocation.Method.ToS
        object rval = invocation.Proceed();
        Console.WriteLine("After: " + invocation.Method.ToS
        return rval;
    }
}
```

The following XML is used to automatically create an AOP proxy and apply a Debug interceptor to object definitions whose names match "English*" and "PortugueseSpeaker".

As with ProxyFactoryObject, there is an InterceptorNames property rather than a list of interceptors, to allow correct behavior for prototype advisors. Named "interceptors" can be advisors or any advice type.

The same advice will be applied to all matching objects. Note that if advisors are used (rather than the interceptor in the above example), the pointcuts may apply differently to different objects.

Running the following simple program demonstrates the application of the AOP interceptor.

The output is shown below

```
ItalianSpeakerOne says; Ciao Mondo!
EnglishSpeakerTwo says; Before: Void SayHello()
Hello World!
After: Void SayHello()
```

```
PortugeseSpeaker says; Before: Void SayHello()
Oi Mundo!
After: Void SayHello()
EnglishSpeakerOne says; Before: Void SayHello()
Hello World!
After: Void SayHello()
```

13.9.1.2. DefaultAdvisorAutoProxyCreator

A more general and extremely powerful auto proxy creator is <code>DefaultAdvisorAutoProxyCreator</code>. This will automatically apply eligible advisors in the current application context, without the need to include specific object names in the autoproxy advisor's object definition. It offers the same merit of consistent configuration and avoidance of duplication as <code>ObjectNameAutoProxyCreator</code>.

Using this mechanism involves:

- Specifying a DefaultAdvisorAutoProxyCreator object definition
- Specifying any number of Advisors in the same or related contexts. Note that these *must* be Advisors, not just interceptors or other advices. This is necessary because there must be a pointcut to evaluate, to check the eligibility of each advice to candidate object definitions.

The **DefaultAdvisorAutoProxyCreator** will automatically evaluate the pointcut contained in each advisor, to see what (if any) advice it should apply to each object defined in the application context.

This means that any number of advisors can be applied automatically to each business object. If no pointcut in any of the advisors matches any method in a business object, the object will not be proxied.

The **DefaultAdvisorAutoProxyCreator** is very useful if you want to apply the same advice consistently to many business objects. Once the infrastructure definitions are in place, you can simply add new business objects without including specific proxy configuration. You can also drop in additional aspects very easily--for example, tracing or performance monitoring aspects-with minimal change to configuration.

The following example demonstrates the use of

DefaultAdvisorAutoProxyCreator. Expanding on the previous example code used to demonstrate ObjectNameAutoProxyCreator we will add a new class, SpeakerDao, that acts as a Data Access Object to find and store IHelloWorldSpeaker objects.

```
public interface ISpeakerDao
{
    IList FindAll();
    IHelloWorldSpeaker Save(IHelloWorldSpeaker speaker);
}
public class SpeakerDao : ISpeakerDao
    public System.Collections.IList FindAll()
    {
        Console.WriteLine("Finding speakers...");
        // just a demo...fake the retrieval.
        Thread.Sleep(10000);
        HelloWorldSpeaker speaker = new HelloWorldSpeaker();
        speaker.Language = Language.Portuguese;
        IList list = new ArrayList();
        list.Add(speaker);
        return list;
    }
    public IHelloWorldSpeaker Save(IHelloWorldSpeaker speake
        Console.WriteLine("Saving speaker...");
        // just a demo...not really saving...
        return speaker;
    }
}
```

The XML configuration specifies two Advisors, that is, the combination of advice (the behavior to add) and a pointcut (where the behavior should be applied). A RegularExpressionMethodPointcutAdvisor is used as a

convenience to specify the pointcut as a regular expression that matches methods names. Other pointcuts of your own creation could be used, in which case a <code>DefaultPointcutAdvisor</code> would be used to define the Advisor. The object definitions for these advisors, advice, and SpeakerDao object are shown below

```
<object id="SpeachAdvisor" type="Spring.Aop.Support.RegularE</pre>
    property name="advice" ref="debugInterceptor"/>
    property name="patterns">
        st>
            <value> *Say *</value>
        </list>
    </property>
</object>
<object id="AdoAdvisor" type="Spring.Aop.Support.RegularExpr</pre>
    property name="advice" ref="timingInterceptor"/>
    property name="patterns">
        st>
            <value>.*Find.*</value>
        </list>
    </property>
</object>
// Advice
<object id="debugInterceptor" type="AopPlay.DebugInterceptor</pre>
<object id="timingInterceptor" type="AopPlay.TimingIntercept</pre>
// Speaker DAO Object - has 'FindAll' Method.
<object id="speakerDao" type="AopPlay.SpeakerDao, AopPlay"/>
// HelloWorldSpeaker objects as previously listed.
```

Adding an instance of DefaultAdvisorAutoProxyCreator to the

```
<object id="ProxyCreator" type="Spring.Aop.Framework.AutoPro</pre>
```

will apply the debug interceptor on all objects in the context that have a method that contains the text "Say" and apply the timing interceptor on objects in the context that have a method that contains the text "Find". Running the following code demonstrates this behavior. Note that the "Save" method of SpeakerDao does not have any advice applied to it.

This produces the following output

```
ItalianSpeakerOne says; Before: Void SayHello()
Ciao Mondo!
After: Void SayHello()
EnglishSpeakerTwo says; Before: Void SayHello()
Hello World!
After: Void SayHello()
PortugeseSpeaker says; Before: Void SayHello()
Oi Mundo!
After:
       Void SayHello()
EnglishSpeakerOne says; Before: Void SayHello()
Hello World!
After: Void SayHello()
Finding speakers...
Elapsed time = 00:00:10.0154745
Saving speaker...
```

The DefaultAdvisorAutoProxyCreator offers support for filtering (using a naming convention so that only certain advisors are evaluated, allowing use of multiple, differently configured, AdvisorAutoProxyCreators in the same factory) and ordering. Advisors can implement the Spring.Core.IOrdered interface to ensure correct ordering if this is an issue. The default is unordered.

13.9.1.3. PointcutFilteringAutoProxyCreator

An AutoProxyCreator that identified objects to proxy by matching a specified IPointcut.

13.9.1.4. TypeNameAutoProxyCreator

An AutoProxyCreator that identifies objects to proxy by matching their Type.FullName against a list of patterns.

13.9.1.5. AttributeAutoProxyCreator

An AutoProxyCreator, that identifies objects to be proxied by checking any System. Attribute defined on a given type and that types interfaces.

13.9.1.6. AbstractFilteringAutoProxyCreator

The base class for AutoProxyCreator implementations that mark objects eligible for proxying based on arbitrary criteria.

13.9.1.7. AbstractAutoProxyCreator

This is the superclass of DefaultAdvisorAutoProxyCreator. You can create your own autoproxy creators by subclassing this class, in the unlikely event that advisor definitions offer insufficient customization to the behavior of the framework DefaultAdvisorAutoProxyCreator.

13.9.2. Using attribute-driven auto-proxying

A particularly important type of autoproxying is driven by attributes. The programming model is similar to using Enterprise Services with ServicedComponents.

In this case, you use the **DefaultAdvisorAutoProxyCreator**, in combination with Advisors that understand attributes. The Advisor pointcut is identified by the presence of .NET attribute in the source code and it is

configured via the data and/or methods of the attribute. This is a powerful alternative to identifying the advisor pointcut and advice configuration through traditional property configuration, either programmatic or through XML based configuration.

Several of the aspect provided with Spring use attribute driven autoproxying. The most prominent example is <u>Transaction</u> support.

13.10. Using AOP Namespace

The AOP namespace allows you to define an advisor, i.e pointcut + 1 piece of advice, in a more declarative manner. Under the covers the DefaultAdvisorAutoProxyCreator is being used. Here is an example,

```
<objects xmlns="http://www.springframework.net"</pre>
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-
                 xmlns:aop="http://www.springframework.net/ad
        <aop:config>
                 <aop:advisor id="getDescriptionAdvisor" poir</pre>
        </aop:config>
        <object id="getDescriptionCalls"</pre>
                 type="Spring.Aop.Support.SdkRegularExpression
                 cproperty name="patterns">
                          st>
                                   <value>.*GetDescription.*</v/>
                          </list>
                 </property>
        </object>
        <object id="getDescriptionCounter" type="Spring.Aop.</pre>
        <object name="testObject" type="Spring.Objects.TestO"</pre>
</objects>
```

In this example, the TestObject, which implements the interface ITestObject, is having AOP advice applied to it. The method GetDescription() is specified as a regular expression pointcut. The aop:config tag and subsequent child tag, aop:advisor, brings together the pointcut with the advice.

In order to have Spring.NET recognise the aop namespace, you need to declare the namespace parser in the main Spring.NET configuration section. For convenience this is shown below. Please refer to the section titled <u>context</u>

configuration for more extensive information..

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.0"</pre>
      <section name="objects" type="Spring.Context.Support.[</pre>
      <section name="parsers" type="Spring.Context.Support.N</pre>
     </sectionGroup>
  </configSections>
  <spring>
    <parsers>
      <parser type="Spring.Aop.Config.AopNamespaceParser, Sp</pre>
    </parsers>
    <context>
      <resource uri="config://spring/objects"/>
    </context>
    <objects xmlns="http://www.springframework.net">
    </objects>
  </spring>
</configuration>
```

13.11. Using TargetSources

Spring.NET offers the concept of a *TargetSource*, expressed in the Spring.Aop.ITargetSource interface. This interface is responsible for returning the "target object" implementing the joinpoint. The TargetSource implementation is asked for a target instance each time the AOP proxy handles a method invocation.

Developers using Spring.NET AOP don't normally need to work directly with TargetSources, but this provides a powerful means of supporting pooling, hot swappable and other sophisticated targets. For example, a pooling TargetSource can return a different target instance for each invocation, using a pool to manage instances.

If you do not specify a TargetSource, a default implementation is used that wraps a local object. The same target is returned for each invocation (as you would expect).

Let's look at the standard target sources provided with Spring.NET, and how you can use them.

When using a custom target source, your target will usually need to be a prototype rather than a singleton object definition. This allows Spring.NET to create a new target instance when required.

13.11.1. Hot swappable target sources

The

org.Spring.NETframework.aop.target.HotSwappable exists to allow the target of an AOP proxy to be switched while allowing callers to keep their references to it.

Changing the target source's target takes effect immediately. The HotSwappableTargetSource is thread safe.

You can change the target via the SWap() method on HotSwappableTargetSource as follows:

```
HotSwappableTargetSource swapper =
      (HotSwappableTargetSource) objectFactory.GetObject("swap
object oldTarget = swapper.swap(newTarget);
```

The XML definitions required look as follows:

The above SWap () call changes the target of the swappable object. Clients who hold a reference to that object will be unaware of the change, but will immediately start hitting the new target.

Although this example doesn't add any advice--and it's not necessary to add advice to use a TargetSource--of course any TargetSource can be used in conjunction with arbitrary advice.

13.11.2. Pooling target sources

Using a pooling target source provides a programming model in which a pool of identical instances is maintained, with method invocations going to free objects in the pool.

A crucial difference between Spring.NET pooling and pooling in .NET Enterprise Services pooling is that Spring.NET pooling can be applied to any PONO. (Plain old .NET object). As with Spring.NET in general, this service can be applied in a non-invasive way.

Spring.NET provides out-of-the-box support using a pooling implementation based on Jakarta Commons Pool 1.1, which provides a fairly efficient pooling implementation. It's also possible to subclass Spring.Aop.Target.AbstractPoolingTargetSource to support any other pooling API.

Sample configuration is shown below:

Note that the target object--"businessObjectTarget" in the example--must be a prototype. This allows the PoolingTargetSource implementation to create new instances of the target to grow the pool as necessary. See the SDK documentation for AbstractPoolingTargetSource and the concrete subclass you wish to use for information about it's properties: maxSize is the most basic, and always guaranteed to be present.

In this case, "myInterceptor" is the name of an interceptor that would need to be defined in the same IoC context. However, it isn't necessary to specify interceptors to use pooling. If you want only pooling, and no other advice, don't set the interceptorNames property at all.

It's possible to configure Spring.NET so as to be able to cast any pooled object to the Spring.Aop.Target.PoolingConfig interface, which exposes information about the configuration and current size of the pool through an introduction. You'll need to define an advisor like this:

This advisor is obtained by calling a convenience method on the

AbstractPoolingTargetSource class, hence the use of MethodInvokingFactoryObject. This advisor's name ('poolConfigAdvisor' here) must be in the list of interceptor names in the ProxyFactoryObject exposing the pooled object.

The cast will look as follows:

```
PoolingConfig conf = (PoolingConfig) objectFactory.GetObject
Console.WriteLine("Max pool size is " + conf.getMaxSize());
```

Pooling stateless service objects is not usually necessary. We don't believe it should be the default choice, as most stateless objects are naturally threadsafe, and instance pooling is problematic if resources are cached.

Simpler pooling is available using autoproxying. It's possible to set the TargetSources used by any autoproxy creator.

13.11.3. Prototype target sources

Setting up a "prototype" target source is similar to a pooling TargetSource. In this case, a new instance of the target will be created on every method invocation. Although the cost of creating a new object may not be high, the cost of wiring up the new object (satisfying its IoC dependencies) may be more expensive. Thus you shouldn't use this approach without very good reason.

To do this, you could modify the **poolTargetSource** definition shown above as follows. (the name of the definition has also been changed, for clarity.)

There is only one property: the name of the target object. Inheritance is used in the TargetSource implementations to ensure consistent naming. As with the pooling target source, the target object must be a prototype object definition, the singleton property of the target should be set to false.

13.11.4. ThreadLocal target sources

ThreadLocal target sources are useful if you need an object to be created for

each incoming request (per thread that is). The concept of a ThreadLocal provides a facility to transparently store resource alongside a thread. Setting up a ThreadLocalTargetSource is pretty much the same as was explained for the other types of target source:

13.12. Defining new Advice types

Spring.NET AOP is designed to be extensible. While the interception implementation strategy is presently used internally, it is possible to support arbitrary advice types in addition to interception around, before, throws, and after returning advice, which are supported out of the box.

The Spring.Aop.Framework.Adapter package is an SPI (Service Provider Interface) package allowing support for new custom advice types to be added without changing the core framework. The only constraint on a custom Advice type is that it must implement the AopAlliance.Aop.IAdvice tag interface.

Please refer to the Spring.Aop.Framework.Adapter namespace documentation for further information.

13.13. Further reading and resources

The Spring.NET team recommends the excellent *AspectJ in Action* by Ramnivas Laddad (Manning, 2003) for an introduction to AOP.

If you are interested in more advanced capabilities of Spring.NET AOP, take a look at the test suite as it illustrates advanced features not discussed in this document.

Chapter 14. Aspect Library

14.1. Introduction

Spring provides several aspects in the distribution. The most popular of which is transactional advice, located in the Spring.Data module. However, the aspects that are documented in this section are those contained within the Spring.Aop module itself. The aspects in within Spring.Aop.dll are Caching, Exception Handling, Logging, Retry, and Parameter Validation. Other traditional advice types such as validation, security, and thread management, will be included in a future release.

14.2. Caching

Caching the return value of a method or the value of a method parameter is a common approach to increase application performance. Application performance is increased with effective use of caching since layers in the application that are closer to the user can return information within their own layer as compared to making more expensive calls to retrieve that information from a lower, and more slow, layer such as a database or a web service. Caching also can help in terms of application scalability, which is generally the more important concern.

The caching support in Spring.NET consists of base cache interfaces that can be used to specify a specific storage implementation of the cache and also an aspect that determines where to apply the caching functionality and its configuration.

The base cache interface that any cache implementation should implement is Spring.Caching.ICache located in Spring.Core. Two implementations are provided, Spring.Caching.AspNetCache located in Spring.Web which stores cache entries within an ASP.NET cache and a simple implementation, Spring.Caching.NonExpiringCache that stores cache entries in memory and never expires these entries. Custom implementations based on 3rd party implementations, such as Oracle Coherence, or memcached, can be used by implementing the ICache interface.

The cache aspect is Spring.Aspects.Cache.CacheAspect located in Spring.Aop. It consists of three pieces of functionality, the ability to cache return values, method parameters, and explicit eviction of an item from the cache. The aspect currently relies on using attributes to specify the pointcut as well as the behavior, much like the transactional aspect. Future versions will allow for external configuration of the behavior so you can apply caching to a code base without needing to use attributes in the code.

The following attributes are available

- CacheResult used to cache the return value
- CacheResultItems used when returning a collection as a return value
- CacheParameter used to cache a method parameter

• InvalidateCache - used to indicate one or more cache items should be invalidated.

Each CacheResult, CacheResultItems, and CacheParameter attributes define the following properties.

- CacheName the name of the cache implementation to use
- **Key** a string representing a Spring Expression Language (SpEL) expression used as the key in the cache.
- **Condition** a SpEL expression that should be evaluated in order to determine whether the item should be cached.
- **TimeToLive** The amount of time an object should remain in the cache (in seconds).

The **InvalidateCache** attribute has properties for the CacheName, the Key as well as the Condition, with the same meanings as listed previously.

Each ICache implementation will have properties that are specific to a caching technology. In the case of AspNetCache, the two important properties to configure are:

- SlidingExperation If this property value is set to true, every time the marked object is accessed it's TimeToLive value is reset to its original value
- **Priority** the cache item priority controlling how likely an object is to be removed from an associated cache when the cache is being purged.
- **TimeToLive** The amount of time an object should remain in the cache (in seconds).

The values of the Priority enumeration are

- LOW low likelihood of deletion when cache is purged.
- **Normal** default priority for deletion when cache is purged.
- **High** high likelihood of deletion when cache is purged.
- NotRemovable cache item not deleted when cache is purged.

An important element of the applying these attributes is the use of the expression language that allows for calling context information to drive the caching actions.

Here is an example taken from the Spring Air sample application of the AirportDao implementation that implements an interface with the method GetAirport(long id).

```
[CacheResult("AspNetCache", "'Airport.Id=' + #id", TimeToLiv
public Airport GetAirport(long id)
{
    // implementation not shown...
}
```

The first parameter is the cache name. The second string parameter is the cache key and is a string expression that incorporates the argument passed into the method, the id. The method parameter names are exposed as variables to the key expression. If you do not specify a key, then all the parameter values will be used to cache the returned value. The expression may also call out to other objects in the Spring container allowing for a more complex key algorithm to be encapsulated. The end result is that the Airport object is cached by id for 60 seconds in a cache named AspNetCache. The TimetoLive property could also have been specified on the configuration of the AspNetCache object.

The configuration to enable the caching aspect is shown below

```
<object" id="CacheAspect" type="Spring.Aspects.Cache.CacheAs</pre>
<object id="AspNetCache" type="Spring.Caching.AspNetCache, $</pre>
  property name="SlidingExpiration" value="true"/>
  property name="Priority" value="Low"/>
  cproperty name="TimeToLive" value="00:02:00"/>
</object>
<!-- Apply aspects to DAOs -->
<object type="Spring.Aop.Framework.AutoProxy.ObjectNameAutoF")</pre>
  property name="ObjectNames">
    st>
      <value>*Dao</value>
    </list>
  </property>
  property name="InterceptorNames">
    st>
      <value>CacheAspect</value>
    </list>
```

</property> </object>

in this example an <code>ObjectNameAutoProxyCreator</code> was used to apply the cache aspect to objects that have Dao in their name. The <code>AspNetCache</code> setting for <code>TimeToLive</code> will override the <code>TimeToLive</code> value set at the method level via the attribute.

14.3. Exception Handling

In some cases existing code can be easily adopted to a simple error handling strategy that can perform one of the following actions

- translations either wrap the thrown exception inside a new one or replace it with a new exception type (no inner exception is set).
- return value the exception is ignored and a return value for the method is provided instead
- swallow the exception is ignored.
- execute Execute an abritrary Spring Expression Language (SpEL expression)

The applicability of general exception handling advice depends greatly on how tangled the code is regarding access to local variables that may form part of the exception. Once you get familiar with the feature set of Spring declarative exception handling advice you should evaluate where it may be effectively applied in your code base. It is worth noting that you can still chain together multiple pieces of exception handling advice allowing you to mix the declarative approach shown in this section with the traditional inheritance based approach, i.e. implementing IThrowsAdvice or IMethodInterceptor.

Declarative exception handling is expressed in the form of a mini-language relevant to the domain at hand, exception handling. This could be referred to as a Domain Specific Language (DSL). Here is a simple example, which should hopefully be self explanatory.

What this is instructing the advice to do is the following bit of code when an ArithmeticException is thrown, throw new System.InvalidOperationException("Wrapped ArithmeticException", e), where e is the original ArithmeticException. The default message, "Wrapped

ArithmethicException" is automatically appended. You may however specify the message used in the newly thrown exception as shown below

```
on exception name ArithmeticException wrap System.InvalidOpe
```

Similarly, if you would rather replace the exception, that is do not nest one inside the other, you can use the following syntax

```
on exception name ArithmeticException replace System.Invalid
or
on exception name ArithmeticException replace System.Invalid
```

Both wrap and replace are special cases of the more general translate action. An example of a translate expression is shown below

```
on exception name ArithmeticException translate new System.I
```

What we see here after the translate keyword is text that will be passed into Spring's expression language (SpEL) for evaluation. Refer to the chapter on the expression language for more details. One important feature of the expression evaluation is the availability of variables relating to the calling context when the exception was thrown. These are

- method the MethodInfo object corresponding to the method that threw the exception
- args the argument array to the method that threw the exception, signature is object[]
- target the AOP target object instance.
- e the thrown exception

You can invoke methods on these variables, prefixed by a '#' in the expression. This gives you the flexibility to call special purpose constructors that can have any piece of information accessible via the above variables, or even other external data through the use of SpEL's ability to reference objects within the Spring container.

You may also choose to 'swallow' the exception or to return a specific return

value, for example

```
on exception name ArithmeticException swallow

or

on exception name ArithmeticException return 12
```

You may also simply log the exception

```
on exception name ArithmeticException, ArgumentException log
```

Here we see that a comma delimited list of exception names can be specified.

The logging is performed using the Commons.Logging library that provides an abstraction over the underlying logging implementation. Logging is currently at the debug level with a logger name of "LogExceptionHandler" The ability to specify these values will be a future enhancement and likely via a syntax resembling a constructor for the action, i.e. log(Debug,"LoggerName").

Multiple exception handling statements can be specified within the list shown above. The processing flow is on exception, the name of the exception listed in the statement is compared to the thrown exception to see if there is a match. A comma separated list of exceptions can be used to group together the same action taken for different exception names. If the action to take is logging, then the logging action is performed and the search for other matching exception names continues. For all other actions, namely translate, wrap, replace, swallow, return, once an exception handler is matched, those in the chain are no longer evaluated. Note, do not confuse this handler chain with the general advice AOP advice chain. For translate, wrap, and replace actions a SpEL expression is created and used to instantiate a new exception (in addition to any other processing that may occur when evaluating the expression) which is then thrown.

The exception handling DSL also supports the ability to provide a SpEL boolean expression to determine if the advice will apply instead of just filtering by the expression name. For example, the following is the equivalent to the first example based on exception names but compares the specific type of the

```
on exception (#e is T(System.ArithmeticException)) wrap Syst
```

The syntax use is 'on exception (SpEL boolean expression)' and inside the expression you have access to the variables of the calling context listed before, i.e. method, args, target, and e. This can be useful to implement a small amount of conditional logic, such as checking for a specific error number in an exception, i.e. (#e is T(System.Data.SqlException) && #e.Errors[0].Number in {156,170,207,208}), to catch and translate bad grammar codes in a SqlException.

While the examples given above are toy examples, they could just as easily be changed to convert your application specific exceptions. If you find yourself pushing the limits of using SpEL expressions, you will likely be better off creating your own custom aspect class instead of a scripting approach.

You can also configure the each of the Handlers individually based on the action keyword. For example, to configure the logging properties on the LogExceptionHandler.

```
<object name="logExceptionHandler" type="Spring.Aspects.Exce"</pre>
  property name="LogName" value="Cms.Session.ExceptionHandl
  property name="LogLevel" value="Debug"/>
  cproperty name="LogMessageOnly" value="true"/>
</object>
<object name="exceptionHandlingAdvice" type="Spring.Aspects.</pre>
  cproperty name="ExceptionHandlerDictionary">
    <dictionary>
      <entry key="log" ref="logExceptionHandler"/>
    </dictionary>
  </property>
  property name="ExceptionHandlers">
    st>
      <value>on exception name ArithmeticException, ArgumentE
    </list>
  </property>
</object>
```

You can also configure ExceptionHandlerAdvice to use an instance of IExceptionHandler by specifing it as an entry in the ExceptionHandlers list. This gives you complete control over all properties of the handler but you must set ConstraintExpressionText and ActionExpressionText which are normally parsed for you from the string. To use the case of configuring the LogExceptionHandler, this approach also lets you specify advanced logging functionality, but at a cost of some additional complexity. For example setting the logging level and pass the exception into the logging subsystem

The configuration of the logger name, level, and weather or not to pass the thrown exception as the second argument to the log method will be supported in the DSL style in a future release.

14.3.1. Language Reference

```
The general syntax of the language is
```

```
on exception name [ExceptionName1, ExceptionName2,...] [action] [SpEL expression]

or
```

on exception (SpEL boolean expression) [action]
[SpEL expression]

The exception names are required as well as the action. The valid actions are

log

- translate
- wrap
- replace
- return
- swallow
- execute

The form of the expression depends on the action. For logging, the entire string is taken as the SpEL expression to log. Translate expects an exception to be returned from evaluation the SpEL expression. Wrap and replace are shorthand for the translate action. For wrap and replace you specify the exception name and the message to pass into the standard exception constructors (string, exception) and (string). The exception name can be a partial or fully qualified name. Spring will attempt to resolve the typename across all referenced assemblies. You may also register type aliases for use with SpEL in the standard manner with Spring.NET and those will be accessible from within the exception handling expression.

14.4. Logging

The logging advice lets you log the information on method entry, exit and thrown exception (if any). The implementation is based on the logging library, Common.Logging, that provides portability across different logging libraries. There are a number of configuration options available, listed below

- LogUniqueIdentifier
- LogExecutionTime
- LogMethodArguments
- LogReturnValue
- Separator
- LogLevel

You declare the logging advice in IoC container with the following XML fragment. Alternatively, you can use the class **SimpleLoggingAdvice** programatically.

```
<object name="loggingAdvice" type="Spring.Aspects.Logging.Si</pre>
  property name="logUniqueIdentifier" value="true"/>
  property name="logExecutionTime"
                                       value="true"/>
  property name="logMethodArguments"
                                       value="true"/>
  property name="LogReturnValue"
                                       value="true"/>
  property name="Separator"
                                       value=";"/>
  property name="LogLevel"
                                       value="Info"/>
  property name="HideProxyTypeNames"
                                       value="true"/>
  property name="UseDynamicLogger"
                                       value="true"/>
</object>
```

The default values for LogUniqueIdentifier, LogExecutionTime, LogMethodArguments and LogReturnValue are false. The default separator value is ", " and the default log level is Common.Logging's LogLevel.Trace.

You can set the name of the logger with the property LoggerName, for example "DataAccessLayer" for a logging advice that would be applied across the all the classes in the data access layer. That works well when using a

'category' style of logging. If you do not set the LoggerName property, then the type name of the logging advice is used as the logging name. Another approach to logging is to log based on the type of the object being called, the target type. Since often this is a proxy class with a relatively meaningless name, the property HideProxyTypeNames can be set to true to show the true target type and not the proxy type.

To further extend the functionality of the SimpleLoggingAdvice you can subclass SimpleLoggingAdvice and override the methods

- string GetEntryMessage(IMethodInvocation invocation, string idString)
- string GetExceptionMessage(IMethodInvocation invocation, Exception e, TimeSpan executionTimeSpan, string idString)
- string GetExitMessage(IMethodInvocation invocation, object returnValue, TimeSpan executionTimeSpan, string idString)

The default implementation to calculate a unique identifier is to use a GUID. behavior by overriding the method String alter this You can CreateUniqueIdentifier(). The SimpleLoggingAdvice class inherits from AbstractLoggingAdvice, which has the abstract InvokeUnderLog(IMethodInvocation object method invocation, ILog log) and you can also override the method GetLoggerForInvocation(IMethodInvocation invocation) to customize the logger instance used for logging. Refer to the **SDK** documentation for details subclassing on more AbstractLoggingAdvice.

As an example of the Logging advice's output, adding the advice to the method

```
public int Bark(string message, int[] luckyNumbers)
{
  return 4;
}
```

And calling $Bark("hello", new int[]{1, 2, 3})$, results in the following output

Entering Bark, 5d2bad47-62cd-435b-8de7-91f12b7f433e, message Exiting Bark, 5d2bad47-62cd-435b-8de7-91f12b7f433e, 30453.12

The method parameters values are obtained using the ToString() method. If you would like to have an alternate implementation, say to view some values in an array, override the method string GetMethodArgumentAsString(IMethodInvocation invocation).

The Spring 1.2 release will have an additional logging advice implementation that leverages the Spring Expression Language to further customize the content of the logging messages via simple configuration using similar syntax to the retry and exception handling advice.

14.5. Retry

When making a distributed call it is often a common requirement to be able to retry the method invocation if there was an exception. Typically the exception will be due to a communication issue that is intermittent and retrying over a period of time will likely result in a successful invocation. When applying retry advice it is important to know if making two calls to the remote service will cause side effects. Generally speaking, the method being invoked should be idempotent, that is, it is safe to call multiple times.

The retry advice is specified using a little language, i.e a DSL. A simple example is shown below

```
on exception name ArithmeticException retry 3x delay 1s
```

The meaning is: when an exception that has 'ArithmeticException' in its type name is thrown, retry the invocation up to 3 times and delay for 1 second between each retry event.

You can also provide a SpEL (Spring Expression Language) expression that calculates the time interval to sleep between each retry event. The syntax for this is shown below

```
on exception name ArithmeticException retry 3x rate (1*#n +
```

As with the exception handling advice, you may also specify a boolean SpEL that must evaluate to true in order for the advice to apply. For example

```
on exception (#e is T(System.ArithmeticException)) retry 3x on exception (#e is T(System.ArithmeticException)) retry 3x
```

The time specified after the delay keyword is converted to a TimeSpan object using Spring's TimeSpanConverter. This supports setting the time as an integer + time unit. Time units are (d, h, m, s, ms) representing (days, hours, minutes, seconds, and milliseconds). For example; 1d = 1day, 5h = 5 hours etc. You can not specify a string such as '1d 5h'. The value that is calculated from the expression after the rate keyword is interpreted as a number of seconds. The

power of using SpEL for the rate expression is that you can easily specify some exponential retry rate (a bigger delay for each retry attempt) or call out to a custom function developed for this purpose.

When using a SpEL expression for the filter condition or for the rate expression, the following variable are available

- method the MethodInfo object corresponding to the method that threw the exception
- args the argument array to the method that threw the exception, signature is object[]
- target the AOP target object instance.
- e the thrown exception

You declare the advice in IoC container with the following XML fragment. Alternatively, you can use the RetryAdvice class programatically.

14.5.1. Language Reference

The general syntax of the language is

```
exception
on
                                           name
[ExceptionName1, ExceptionName2, ...]
                                          retry
          of times]x [delay|rate]
Inumber
                                         [delay
time|SpEL rate expression]
or
              (SpEL boolean expression)
   exception
                                          retry
on
[number
          of times]x
                         [delay|rate]
                                         [delay
time|SpELrate expression]
```

14.6. Transactions

The transaction aspect is more fully described in the section on <u>transaction</u> <u>management</u>.

14.7. Parameter Validation

Spring provides a UI-agnostic <u>validation framework</u> in which you can declare validation rules, both progammatically and declaratively, and have those rules evaluated against an arbitrary .NET object. Spring provides additional support for the rendering of validation errors within Spring's ASP.NET framework. (See the section on <u>ASP.NET usage tips</u> for more information.) However, validation is not confined to the UI tier. It is a common task that occurs across most, if not all, applications layers. Validation that is performed in the UI layer is often repeated in the service layer, in order to be proactive in case non UI-based clients invoke the service layer. Validation rules completely different from those used in the UI layer may also be used on the server side.

To address some of the common needs for validation on the server side, Spring provides parameter validation advice so that applies Spring's validation rules to the method parameters. The class <code>ParameterValidationAdvice</code> is used in conjunction with the <code>Validated</code> attribute to specify which validation rules are applied to method parameters. For example, to apply parameter validation to the method SuggestFlights in the BookingAgent class used in the <code>SpringAir sample application</code>, you would apply the <code>Validated</code> attribute to the method parameters as shown below.

```
public FlightSuggestions SuggestFlights( [Validated("tripVal
{
    // unmodified implementation goes here
}
```

The Validated attribute takes a string name that specifies the name of the validation rule, i.e. the name of the IValidator object in the Spring application context. The Validated attribute is located in the namespace Spring.Validation of the Spring.Core assembly.

The configuration of the advice is to simply define the an instance of the ParameterValidationAdvice class and apply the advice, for example based on object names using an ObjectNameAutoProxyCreator, as shown below,

```
<object id="validationAdvice" type="Spring.Aspects.Validation")</pre>
```

When the advised method is invoked first the validation of each method parameter is performed. If all validation succeeds, then the method body is executed. If validation fails an exception of the type ValidationException is thrown and you can retrieve errors information from its property ValidationErrors. See the SDK documentation for details.

Chapter 15. Common Logging

15.1. Introduction

Spring uses a simple logging abstraction in order to provide a layer of indirection between logging calls made by Spring and the specific logging library used in your application (log4net, EntLib logging, NLog). The library is available for .NET 1.0, 1.1, and 2.0 with both debug and strongly signed assemblies. Since this need is not specific to Spring, the logging library was moved out of the Spring project and into a more general open source project called Common Infrastructure Libraries for .NET. The logging abstraction within the project is known as Common.Logging. Note that it is not the intention of this library to be a replacement for the many fine logging libraries that are out there. The API is incredibly minimal and will very likely stay that way. Please note that this library is intended only for use where the paramount requirement is portability and you will generally be better served by using a specific logging implementation so that you can leverage its advanced features and extended APIs to your advantage.

You can find online documentation on how to configure Common.Logging is available in https://example.com/html/help formats.

Chapter 16. Testing

16.1. Introduction

The Spring team considers developer testing to be an absolutely integral part of enterprise software development. A thorough treatment of testing in the enterprise is beyond the scope of this chapter; rather, the focus here is on the value add that the adoption of the IoC principle can bring to <u>unit testing</u>; and on the benefits that the Spring Framework provides in <u>integration testing</u>.

16.2. Unit testing

One of the main benefits of Dependency Injection is that your code is much less likely to have any hidden dependencies on the runtime environment or other configuration subsystems. This allows for unit tests to be written in a manner such that the object under test can be simply instantiated with the **new** operator and have its dependences set in the unit test code. You can use mock objects (in conjunction with many other valuable testing techniques) to test your code in isolation. If you follow the architecture recommendations around Spring you will find that the resulting clean layering and componentization of your codebase will naturally faciliate *easier* unit testing. For example, you will be able to test service layer objects by stubbing or mocking DAO interfaces, without any need to access persistent data while running unit tests.

True unit tests typically will run extremely quickly, as there is no runtime infrastructure to set up, i.e., database, ORM tool, or whatever. Thus emphasizing true unit tests as part of your development methodology will boost your productivity. The upshot of this is that you do not need this section of the testing chapter to help you write effective *unit* tests for your IoC-based applications.

16.3. Integration testing

However, it is also important to be able to perform some integration testing enabling you to test things such as:

- The correct wiring of your Spring IoC container contexts.
- Data access using ADO.NET or an ORM tool. This would include such things such as the correctness of SQL statements / or NHibernate XML mapping files.

The Spring Framework provides first class support for integration testing in the form of the classes that are packaged in the Spring. Testing. NUnit.dll library. Please note that these test classes are NUnit-specific. Support for mbUnit and VSTS are under consideration for future versions.



Note

The Spring.Testing.NUnit.dll library is compiled against NUnit 2.4.1. At the tim of this writing the latest version of NUnit is 2.4.6. Note that add-in have their own versions of NUnit they use. For example, ReSharper 3.0 uses 2.2.8. If you are using the GUI-runner that comes with NUnit then you should add the following to your .config file, (in the form of MyAssembly.dll.config)

The Spring. Testing. NUnit namespace provides valuable NUnit

TestCase superclasses for integration testing using a Spring container. Note that as of NUnit 2.4 these can be rewritten in terms of custom attributes via NUnit's new extensibility mechanism. This will be an additional option in an upcoming release of Spring.NET and is already present in the Java version of the Spring framework.

These superclasses provide the following functionality:

- Spring IoC container caching between test case execution.
- The pretty-much-transparent <u>Dependency Injection of test fixture instances</u> (this is nice).
- <u>Transaction management</u> appropriate to integration testing (this is even nicer).
- A number of Spring-specific <u>inherited instance variables</u> that are really useful when integration testing.

16.3.1. Context management and caching

The Spring.Testing.NUnit package provides support for consistent loading of Spring contexts, and caching of loaded contexts. Support for the caching of loaded contexts is important, because if you are working on a large project, startup time may become an issue - not because of the overhead of Spring itself, but because the objects instantiated by the Spring container will themselves take time to instantiate. For example, a project with 50-100 NHibernate mapping files might take 10-20 seconds to load the mapping files, and incurring that cost before running every single test case in every single test fixture will lead to slower overall test runs that could reduce productivity.

To address this issue, the AbstractDependencyInjectionSpringContextTests has an protected property that subclasses must implement to provide the location of context definition files:

```
protected abstract string[] ConfigLocations { get; }
```

Implementations of this method must provide an array containing the IResource locations of XML configuration metadata used to configure the application. This will be the same, or nearly the same, as the list of configuration locations specified in App.config/Web.config or other deployment

configuration.

By default, once loaded, the configuration file set will be reused for each test case. Thus the setup cost will be incurred only once (per test fixture), and subsequent test execution will be much faster. In the unlikely case that a test may 'dirty' the config location, requiring reloading - for example, by changing an object definition or the state of an application object - you can call the SetDirty() method on

AbstractDependencyInjectionSpringContextTests to cause the test fixture to reload the configurations and rebuild the application context before executing the next test case.

16.3.2. Dependency Injection of test fixtures

When

AbstractDependencyInjectionSpringContextTests (and subclasses) load your application context, they can optionally configure instances of your test classes by Setter Injection. All you need to do is to define instance variables and the corresponding setters. AbstractDependencyInjectionSpringContextTests will automatically locate the corresponding object in the set of configuration files specified in the ConfigLocationS property.

Consider the scenario where we have a class, **HibernateTitleDao**, that performs data access logic for say, the **Title** domain object. We want to write integration tests that test all of the following areas:

- The Spring configuration; basically, is everything related to the configuration of the HibernateTitleDao object correct and present?
- The Hibernate mapping file configuration; is everything mapped correctly and are the correct lazy-loading settings in place?
- The logic of the **HibernateTitleDao**; does the configured instance of this class perform as anticipated?

Let's look at the test class itself (we will look at the configuration immediately afterwards).

[TestFixture]

```
public class HibernateTitleDaoTests : AbstractDependencyInje
    // this instance will be (automatically) dependency inje
    private HibernateTitleDao titleDao;
    // a setter method to enable DI of the 'titleDao' instar
    public HibernateTitleDao HibernateTitleDao {
        set { titleDao = value; }
    }
    [Test]
    public void LoadTitle() {
        Title title = this.titleDao.LoadTitle(10);
        Assert.IsNotNull(title);
    }
    // specifies the Spring configuration to load for this t
    protected override string[] ConfigLocations {
        return new String[] { "assembly://MyAssembly/MyNames
    }
}
```

The file referenced by the ConfigLocations method ('classpath:com/foo/daos.xml') looks like this:

AbstractDependencyInjectionSpringContextTests

classes uses <u>autowire by type</u>. Thus if you have multiple object definitions of the same type, you cannot rely on this approach for those particular object. In that case, you can use the inherited applicationContext instance variable, and explicit lookup using (for example) an explicit call to applicationContext.GetObject("titleDao").

If you don't want dependency injection applied to your test cases, simply don't declare any set properties. Alternatively, you can extend the AbstractSpringContextTests - the root of the class hierarchy in the Spring.Testing.NUnit namespace. It merely contains convenience methods to load Spring contexts, and performs no Dependency Injection of the test fixture.

16.3.2.1. Field level injection

If, for whatever reason, you don't fancy having setter properties in your test fixtures, Spring can (in this one case) inject dependencies into protected fields. Find below a reworking of the previous example to use field level injection (the Spring XML configuration does not need to change, merely the test fixture).

```
protected override string[] ConfigLocations {
    return new String[] { "assembly://MyAssembly/MyNames
}
```

In the case of field injection, there is no autowiring going on: the name of your protected instances variable(s) are used as the lookup object name in the configured Spring container.

16.3.3. Transaction management

One common issue in tests that access a real database is their effect on the state of the persistence store. Even when you're using a development database, changes to the state may affect future tests. Also, many operations - such as inserting to or modifying persistent data - cannot be done (or verified) outside a transaction.

The

AbstractTransactionalDbProviderSpringContextTes superclass (and subclasses) exist to meet this need. By default, they create and roll back a transaction for each test. You simply write code that can assume the existence of a transaction. If you call transactionally proxied objects in your tests, they will behave correctly, according to their transactional semantics.

AbstractTransactionalSpringContextTests depends on a IPlatformTransactionManager object being defined in the application context. The name doesn't matter, due to the use of autowire by type.

Typically you will extend the subclass, AbstractTransactionalDbProviderSpringContextTes This also requires that a DbProvider object definition - again, with any name - be present in the configurations. It creates an AdoTemplate instance variable that is useful for convenient querying, and provides handy methods to delete the contents of selected tables (remember that the transaction will roll back by default, so this is safe to do).

If you want a transaction to commit - unusual, but occasionally useful when you want a particular test to populate the database - you can call the SetComplete() method inherited from

AbstractTransactionalSpringContextTests. This will cause the transaction to commit instead of roll back.

There is also convenient ability to end a transaction before the test case ends, through calling the <code>EndTransaction()</code> method. This will roll back the transaction by default, and commit it only if <code>SetComplete()</code> had previously been called. This functionality is useful if you want to test the behavior of 'disconnected' data objects, such as Hibernate-mapped objects that will be used in a web or remoting tier outside a transaction. Often, lazy loading errors are discovered only through UI testing; if you call <code>EndTransaction()</code> you can ensure correct operation of the UI through your NUnit test suite.

16.3.4. Convenience variables

When you extend the AbstractTransactionalDbProviderSpringContextTes class you will have access to the following protected instance variables:

- applicationContext
 IConfigurableApplicationContext): inherited from the
 AbstractDependencyInjectionSpringContextTest:
 superclass. Use this to perform explicit object lookup, or test the state of the
 context as a whole.
- adoTemplate: inherited from AbstractTransactionalDbProviderSpringContext Useful for querying to confirm state. For example, you might query before and after testing application code that creates an object and persists it using an ORM tool, to verify that the data appears in the database. (Spring will ensure that the query runs in the scope of the same transaction.) You will need to tell your ORM tool to 'flush' its changes for this to work correctly, for example using the Flush() method on NHibernate's ISession interface.

Often you will provide an application-wide superclass for integration tests that provides further useful instance variables used in many tests

16.4. Further Resources

This section contains links to further resources about testing in general.

• The NUnit homepage. The Spring Framework's unit test suite is written using NUnit as the testing framework.

Spring Framework



Part II. Middle Tier Data Access

This part of the reference documentation is concerned with othe middle tier, and specifically the data access responsibilities of said tier.

Spring's comprehensive transaction management support is covered in some detail, followed by thorough coverage of the various middle tier data access frameworks and technologies that the Spring Framework integrates with.

- Chapter 17, Transaction management
- Chapter 18, DAO support
- Chapter 19, DbProvider
- Chapter 20, Data access using ADO.NET
- Chapter 21, Object Relational Mapping (ORM) data access

Spring Framework



Chapter 17. Transaction management

17.1. Introduction

Spring.NET provides a consistent abstraction for transaction management that provides the following benefits

- Provides a consistent programming model across different transaction APIs such as ADO.NET, Enterprise Services, System.Transactions, and NHibernate.
- Support for <u>declarative transaction management</u> with any of the above data access technologies
- Provides a simple API for programmatic transaction management
- Integrates with Spring's high level persistence integration APIs such as AdoTemplate.

This chapter is divided up into a number of sections, each detailing one of the value-adds or technologies of the Spring Framework's transaction support. The chapter closes with some discussion of best practices surrounding transaction management.

- The first section, entitled <u>Motivations</u> describes why one would want to use the Spring Framework's transaction abstraction as opposed to using System.Transactions or a specific data access technology transaction API.
- The second section, entitled <u>Key Abstractions</u> outline the core classes as well as how to configure them.
- Th third section, entitled <u>Declarative transaction management</u>, covers support for declarative transaction management.
- The fourth section, entitled <u>Programmatic transaction management</u>, covers support for programmatic transaction management.

17.2. Motivations

The data access technology landscape is a broad one, within the .NET BCL there are three APIs for performing transaction management, namely ADO.NET, Enterprise Services, and System.Transactions. Other data access technologies such as object relational mappers and result-set mapping libraries are also gaining in popularity and each come with their own APIs for transaction management. As such, code is often directly tied to a particular transaction API which means you must make an up-front decision which API to use in your application. Furthermore, if the need arises to change your approach, it quite often will not be a simple refactoring. Using Spring's transaction API you can keep the same API across different data access technologies. Changing the underlying transaction implementation that is used is a simple matter of configuration or a centralized programmatic change as compared to a major overhauling.

Hand in hand with the variety of options available is the establishment generally agreed upon best practices for data access. Martin Fowler's book, Patterns of Enterprise Application Architecture, is an excellent source of approaches to data access that have been successful in the real world. One approach that is quite common is to introduce a data access layer into your architecture. The data access layer is concerned not only with providing some portability between different data access technologies and databases but its scope is strictly related to data access. A simple data access layer would be not much more than data access objects (DAOs) with 'Create/Retrieve/Update/Delete' (CRUD) methods devoid of any business logic. Business logic resides in another application layer, the business service layer, in which business logic will call one or more DAOs to fulfill a higher level end-user function.

In order to perform this end-user function with all-or-nothing transactional semantics, the transaction context is controlled by the business service layer (or other 'higher' layers). In such a common scenario, an important implementation detail is how to make the DAO objects aware of the 'outer' transaction started in another layer. A simplistic implementation of a DAO would perform its own connection and transaction management, but this would not allow grouping of DAO operations with the same transaction as the DAO is doing its own transaction/resource management. As such there needs to be a means to transfer the connection/transaction pair managed in the business service layer to the

DAOs. There are a variety of ways to do this, the most invasive being the explicitly pass a connection/transaction object as method arguments to your DAOs. Another way is to store the connection/transaction pair in thread local storage. In either case, if you are using ADO.NET you must invent some infrastructure code to perform this task.

But wait, doesn't Enterprise Services solve this problem - and what about the functionality in the System. Transactions namespace? The answer is yes...and no. Enterprise Services lets you use the 'raw' ADO.NET API within a transaction context such that multiple DAO operations are grouped within the same transaction. The downside to Enterprise Services is that it always uses distributed (global) transactions via the Microsoft Distributed Transaction Coordinator (MS-DTC). For most applications this is overkill just to get this functionality as global transactions are significantly less performant than local ADO.NET transactions.

There are similar issues with using the 'using TransactionScope' construct within the new System. Transactions namespace. The goal with TransactionScope is to define a, well - transaction scope - within a using statement. Plain ADO.NET code within that using block will then be a local ADO.NET based transaction if only a single transactional resource is accessed. However, the 'magic' of System. Transactions (and the database) is that local transactions will be promoted to distributed transactions when a second transaction resource is detected. The name that this goes by is Promotable Single Phase Enlistment (PSPE). However, there is a big caveat - opening up a second IDbConnection object to the same database with the same database string will trigger promotion from local to global transactions. As such, if your DAOs are performing their own connection management you will end up being bumped up to a distributed transaction. In order to avoid this situation for the common case of an application using a single database, you must pass around a connection object to your DAOs. It is also worth to note that many database providers (Oracle for sure) do not yet support PSPE and as such will always use a distributed transaction even if there is only a single database.

Last but not least is the ability to use declarative transaction management. Not many topics in database transaction-land give developers as much 'bang-for-the-buck' as declarative transactions since the noisy tedious bits of transactional API code in your application are pushed to the edges, usually in the form of class/method attributes. Only Enterprise Services offers this feature in the BCL.

Spring fills the gap - it provides declarative transaction management if you are using local ADO.NET or System.Transactions (the most popular) or other data access technologies. Enterprise Services is not without it small warts as well, such as the need to separate your query/retrieve operations from your create/update/delete operations if you want to use different isolation levels since declarative transaction metadata can only be applied at the class level. Nevertheless, all in all, Enterprise Services, in particular with the new 'Services Without Components' implementation for XP SP2/Server 2003, and hosted within the same process as your application code is as good as it gets out of the .NET box. Despite these positive points, it hasn't gained a significant mindshare in the development community.

Spring's transaction support aims to relieve these 'pain-points' using the data access technologies within the BCL - and for other third party data access technologies as well. It provides declarative transaction management with a configurable means to obtain transaction option metadata - out of the box attributes and XML within Spring's IoC configuration file are supported.

Finally, Spring's transaction support lets you mix data access technologies within a single transaction - for example ADO.NET and NHibernate operations.

With this long winded touchy/feely motivational section behind us, lets move on to see the code.

17.3. Key Abstractions

The key to the Spring transaction management abstraction is the notion of a *transaction strategy*. A transaction strategy is defined by the Spring.Transaction.IPlatformTransactionManager interface, shown below:

```
public interface IPlatformTransactionManager {
   ITransactionStatus GetTransaction( ITransactionDefinition
   void Commit( ITransactionStatus transactionStatus );
   void Rollback( ITransactionStatus transactionStatus );
}
```

This is primarily a 'SPI' (Service Provider Interface), although it can be used Programatically. Note that in keeping with the Spring Framework's philosophy, IPlatformTransactionManager is an interface, and can thus be easily mocked or stubbed as necessary. IPlatformTransactionManager implementations are defined like any other object in the IoC container. The following implementations are provided

- AdoPlatformTransactionManager local ADO.NET based transactions
- ServiceDomainPlatformTransactionManager
 distributed transaction manager from Enterprise Services
- TxScopePlatformTransactionManager local/distributed transaction manager from System.Transactions.
- HibernatePlatformTransactionManager local transaction manager for use with NHibernate or mixed ADO.NET/NHibernate data access operations.

Under the covers, the following API calls are made. For the AdoPlatformTransactionManager, Transaction.Begin(), Commit(), Rollback(). ServiceDomainPlatformTransactionManager uses the 'Services without

that your objects do not need to inherit from Components' update so ServicedComponent or directly call the Enterprise Services ServiceDomain.Enter(), Leave: ContextUtil.SetAbort(). TxScopePlatformTransactionManager calls: TransactionScope(); new Transaction.Current.Rollback(). .Complete(), Dispose(), Configuration properties for each transaction manager are specific to the data access technology used. Refer to the API docs for comprehensive information but the you a should give good basis for getting examples HibernatePlatformTransactionManager is described more in the following section

.

GetTransaction(..) The method returns a **ITransactionStatus** object, depending on a ITransactionDefinition The parameters. returned ITransactionStatus might represent a new or existing transaction (if there was a matching transaction in the current call stack - with the implication being that a ITransactionStatus is associated with a logical thread of execution.

The ITransactionDefinition interface specified

- Isolation: the degree of isolation this transaction has from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?
- Propagation: normally all code executed within a transaction scope will run
 in that transaction. However, there are several options specifying behavior
 if a transactional method is executed when a transaction context already
 exists: for example, simply continue running in the existing transaction (the
 common case); or suspending the existing transaction and creating a new
 transaction.
- Timeout: how long this transaction may run before timing out (and automatically being rolled back by the underlying transaction infrastructure).
- Read-only status: a read-only transaction does not modify any data. Read-only transactions can be a useful optimization in some cases (such as when using NHibernate).

These settings reflect standard transactional concepts. If necessary, please refer

to a resource discussing transaction isolation levels and other core transaction concepts because understanding such core concepts is essential to using the Spring Framework or indeed any other transaction management solution.

The **ITransactionStatus** interface provides a simple way for transactional code to control transaction execution and query transaction status.

Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct IPlatformTransactionManager implementation is absolutely essential. In good Spring fashion, this important definition typically is made using via Dependency Injection.

IPlatformTransactionManager implementations normally require knowledge of the environment in which they work, ADO.NET, NHibernate, etc. The following example shows how a standard ADO.NET based IPlatformTransactionManager can be defined.

We must define a Spring IDbProvider and then use Spring's AdoPlatformTransactionManager, giving it a reference to the IDbProvider. For more information on the IDbProvider abstraction refer to the next chapter.

We can also use a transaction manager based on System. Transactions just as

easily, as shown in the following example

Similarly for the HibernateTransactionManager as shown in the section on ORM transaction management.

Note that in all these cases, application code will not need to change at all since, dependency injection is a perfect companion to using the strategy pattern. We can now change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

17.4. Resource synchronization with transactions

How does application code participate with the resources (i.e. Connection/Transactions/Sessions) that are created/reused/cleanedup via the different transaction managers? There are two approaches - a high-level and a low-level approach

17.4.1. High-level approach

These do not replace native APIs, but internally handle resource creation/reuse, cleanup, and optional transaction synchronization (i.e. event notification) of the resources and exception mapping so that user data access code doesn't have to worry about these concerns at all, but can concentrate purely on non-boilerplate persistence logic. Generally, the same inversion of control approach is used for all persistence APIs. In this approach the API has a callback method or delegate that presents the user code with the relevant resource ready to use - i.e. a DbCommand with its Connection and Transaction properties set based on the transaction option metadata. These classes go by the naming scheme 'template', examples of which are AdoTemplate and HibernateTemplate. Convenient 'one-liner' helper methods in these template classes build upon the core callback/IoC design by providing specific implementations of the callback interface.

17.4.2. Low-level approach

A utility class can be used to directly obtain a connection/transaction pair that is aware of the transactional calling context and returns a pair suitable for that context. The class <code>ConnectionUtils</code> contains the static method <code>ConnectionTxPair</code>

GetConnectionTxPair(IDbProvider provider) which serves this purpose.

17.5. Declarative transaction management

Most Spring users choose declarative transaction management. It is the option with the least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container.

Spring's declarative transaction management is made possible with Spring AOP, although, as the transactional aspects code comes with Spring and may be used in a boilerplate fashion, AOP concepts do not generally have to be understood to make effective use of this code.

The basic approach is to specify transaction behavior (or lack of it) down to the individual method level. It is also possible to mark a transaction for rollback by setting the 'RollbackOnly' property on the ITransactionStatus object returned from the IPlatformTransactionManager within a transaction context if necessary. Some of the highlights of Spring's declarative transaction management are:

- Declarative Transaction management works in any environment. It can work with ADO.NET, System.Transactions, NHibernate etc, with configuration changes only.
- Enables declarative transaction management to be applied to any class, not merely special classes such as those that inherit from ServicedComponent or other infrastructure related base classes.
- Declarative rollback rules. Rollback rules can be control declaratively and allow for only specified exceptions thrown within a transactional context to trigger a rollback
- Spring gives you an opportunity to customize transactional behavior, using AOP. For example if you want to insert custom behavior in the case of a transaction rollback, you can. You can also add arbitrary advice, along with the transactional advice.
- Spring does not support propagation of transaction context across remote calls.

Note rollback rules as configured from XML are still under development.

The concept of rollback rules is important: they enable us to specify which exceptions should cause automatic roll back. We specify this declaratively, in configuration, not in code. So, while we can still set RollbackOnly on the ITransactionStatus object to roll the current transaction back

Programatically, most often we can specify a rule that MyApplicationException must always result in rollback. This has the significant advantage that business objects don't need to depend on the transaction infrastructure. For example, they typically don't need to import any Spring APIs, transaction or other. If you would like to rollback the transaction programmatically and you are using declarative transaction management, use the utility method

TransactionInterceptor.CurrentTransactionStatus.SetRollback(



Prior to Spring.NET 1.2 RC1 the API call would be TransactionInterceptor.CurrentTransactionStatus. = true;

17.5.1. Understanding Spring's declarative transaction implementation

The aim of this section is to dispel the mystique that is sometimes associated with the use of declarative transactions. It is all very well for this reference documentation to simply tell you to annotate your classes with the Transaction attribute and add some boilerplate XML to your IoC configuration, and then expect you to understand how it all works. This section will explain the inner workings of Spring's declarative transaction infrastructure to help you navigate your way back upstream to calmer waters in the event of transaction-related issues.



Looking at the Spring source code is a good way to get a real understanding of Spring's transaction support. You should find the API documentation informative and complete. We suggest turning the logging level to 'DEBUG' in your Spring-enabled application(s) during development to better see what goes on under the hood.

The most important concepts to grasp with regard to Spring's declarative transaction support are that this support is enabled via AOP proxies, and that the transactional advice is driven by metadata (currently XML- or attribute-based). The combination of a proxy with transactional metadata yields an AOP proxy that uses a TransactionInterceptor in conjunction with an

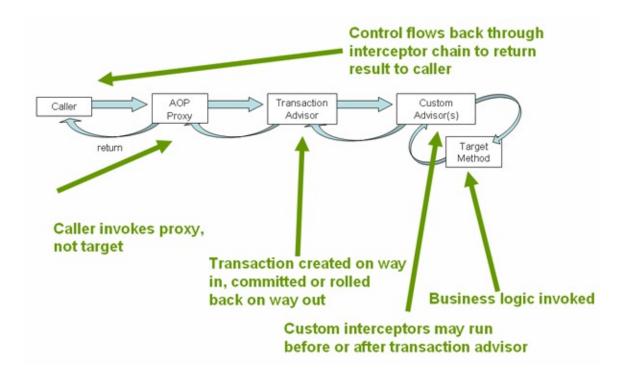
appropriate IPlatformTransactionManager implementation to drive transactions around method invocations.



Note

Although knowledge of AOP (and specifically Spring AOP) is not required in order to use Spring's declarative transaction support, it can help. Spring AOP is thoroughly covered in the AOP chapter.

Conceptually, calling a method on a transactional proxy looks like this.



The flow of events is the following. First the set of objects you would like to apply AOP transactional advice to are identified. There are a variety of ways to configure the Spring IoC container to create proxies for the defined object definitions. The standard Spring AOP based options are

- ProxyFactoryObject. The common properties to set are the reference to the object to proxy (the target object) and a reference to the transaction advice. See <u>Section 13.5</u>, "Using the ProxyFactoryObject to create AOP proxies" for more details.
- AutoProxy Defines criteria to select a collection of objects to create a transactional AOP proxy.

The AutoProxy options are

- ObjectNameAutoProxyCreator which specifies collection of object names based on wildcard matching of object names. See Section 13.9.1.1, "ObjectNameAutoProxyCreator"
- DefaultAdvisorAutoProxyCreator which specifies one or more "advisors" i.e an object representing an aspect, including both an advice and a pointcut targeting it to specific joinpoints. See Section 13.9.1.2, "DefaultAdvisorAutoProxyCreator"

There is also a convenience subclass of ProxyFactoryObject, namely TransactionProxyFactoryObject, that sets some common default values for the specific case of applying transactional advice.

The DefaultAdvisorAutoProxyCreator is very powerful and is the means by which Spring can be configured to use attributes to identify the pointcuts where transaction advice should be applied. The advisor that performs that task is TransactionAttributeSourceAdvisor.



Note, think of the word 'Attribute' in this class name not as the .NET attribute but as the transaction 'options' you want to specify. This name is inherited from the Java version and the name will be changed in the RC1 release to avoid confusion since a common naming convention when creating classes are .NET attributes is to put the word 'Attribute' in the name.

Which one of the many options available should you choose for your development? That depends, each one has it own set of pro's and con's which will be discussed in turn in the following sections.

With the transactional AOP proxy now created we can discuss the flow of events in the code as proxied methods are invoked. When the method is invoked, before calling the target object's method, a transaction is created if one hasn't already been created. Then the target method is invoked. If there was an exception throw, the transaction is typically rolled back, but it can also be committed if the exception type specified in the transaction option, NoRollbackFor, matches the thrown exception. If no exception was thrown, that is taken as a sign of success and the transaction is committed.

When using other AOP advice with the transactional advice you can set the order

of the 'interceptor chain' so that, for example, performance monitoring advice always precede the transactional advice.

17.5.2. A First Example

Consider the following interface. The intent is to convey the concepts to you so you can concentrate on the transaction usage and not have to worry about domain specific details. The <code>ITestObjectManager</code> is a poor-mans business service layer - the implementation of which will make two DAO calls. Clearly this example is overly simplistic from the service layer perspective as there isn't any business logic at all!. The 'service' interface is shown below.

```
public interface ITestObjectManager
{
  void SaveTwoTestObjects(TestObject to1, TestObject to2);
  void DeleteTwoTestObjects(string name1, string name2);
}
```

The implementation of ITestObjectManager is shown below

```
public class TestObjectManager : ITestObjectManager
{
    // Fields/Properties ommited

    [Transaction()]
    public void SaveTwoTestObjects(TestObject to1, TestObject {
        TestObjectDao.Create(to1.Name, to1.Age);
        TestObjectDao.Create(to2.Name, to1.Age);
    }

    [Transaction()]
    public void DeleteTwoTestObjects(string name1, string nam {
        TestObjectDao.Delete(name1);
        TestObjectDao.Delete(name2);
    }
}
```

Note the Transaction attribute on the methods. Other options such as isolation level can also be specified but in this example the default settings are used. However, please note that the mere presence of the Transaction attribute is not enough to actually turn on the transactional behavior - the Transaction attribute is simply metadata that can be consumed by something that is Transaction attribute-aware and that can use the said metadata to configure the appropriate objects with transactional behavior.

The TestObjectDao property has basic create update delete and find method for the 'domain' object TestObject. TestObject in turn has simple properties like name and age.

```
public interface ITestObjectDao
{
  void Create(string name, int age);
  void Update(TestObject to);
  void Delete(string name);
  TestObject FindByName(string name);
  IList FindAll();
}
```

The Create and Delete method implementation is shown below. Note that this uses the AdoTemplate class discussed in the following chapter. Refer to Section 17.4, "Resource synchronization with transactions" for information on the interaction between Spring's high level persistence integration APIs and transaction management features.

```
name));
}
}
```

The TestObjectManager is configured with the DAO objects by standard dependency injection techniques. The client code, which in this case asks directly the Spring IoC container for an instance ITestObjectManager, will receive a transaction proxy with transaction options based on the attribute metadata. Note that typically ITestObjectManager would be set on yet another higher level object via dependency injection, for example a web service.

The client calling code is shown below

```
IApplicationContext ctx =
    new XmlApplicationContext("assembly://Spring.Data.Integra
ITestObjectManager mgr = ctx["testObjectManager"] as ITestOb

TestObject to1 = new TestObject();
to1.Name = "Jack";
to1.Age = 7;

TestObject to2 = new TestObject();
to2.Name = "Jill";
to2.Age = 8;

mgr.SaveTwoTestObjects(to1, to2);

mgr.DeleteTwoTestObjects("Jack", "Jill");
```

The configuration of the object definitions of the DAO and manager classes is shown below.

```
connectionString="Data Source=(local);Data
    <object id="transactionManager"</pre>
            type="Spring.Data.AdoPlatformTransactionManager,
        cproperty name="DbProvider" ref="DbProvider"/>
    </object>
    <object id="adoTemplate" type="Spring.Data.AdoTemplate,</pre>
        property name="DbProvider" ref="DbProvider"/>
    </object>
    <object id="testObjectDao" type="Spring.Data.TestObject[</pre>
        cproperty name="AdoTemplate" ref="adoTemplate"/>
    </object>
    <!-- The object that performs multiple data access opera
    <object id="testObjectManager"</pre>
            type="Spring.Data.TestObjectManager, Spring.Data
        cproperty name="TestObjectDao" ref="testObjectDao"/>
    </object>
</objects>
```

This is standard Spring configuration and as such provides you with the flexibility to parameterize your connection string and to easily switch implementations of your DAO objects. The configuration to create a transactional proxy for the manager class is shown below.

Granted this is a bit verbose and hard to grok at first sight - however you only need to grok this once as it is 'boiler plate' XML you can reuse across multiple projects. What these object definitions are doing is to instruct Spring's to look for all objects within the IoC configuration that have the [Transaction] attribute and then apply the AOP transaction interceptor to them based on the transaction options contained in the attribute. The attribute serves both as a pointcut and as the declaration of transactional option information.

Since this XML fragment is not tied to any specific object references it can be included in its own file and then imported via the <import> element. In examples and test code this XML configuration fragment is named autoDeclarativeServices.xml See Section 5.2.2.1, "Composing XML-based configuration metadata" for more information.

The classes and their roles in this configuration fragment are listed below

- TransactionInterceptor is the AOP advice responsible for performing transaction management functionality.
- TransactionAttributeSourceAdvisor is an AOP Advisor that holds the TransactionInterceptor, which is the advice, and a pointcut (where to apply the advice), in the form of a TransactionAttributeSource.
- AttributesTransactionAttributeSource is an implementation of the ITransactionAttributeSource interface that defines where to get the transaction metadata defining the transaction semantics (isolation level, propagation behavior, etc) that should be applied to specific methods of specific classes. The transaction metadata

implementations specified the is via of ITransactionAttributeSource interface. This example shows the of the implementation use Spring.Transaction.Interceptor.AttributesTra to obtain that information from standard .NET attributes. By the very nature of using standard .NET attributes, the attribute serves double duty in identifying the methods where the transaction semantics apply. Alternative implementations of ITransactionAttributeSource available MatchAlwaysTransactionAttributeSource, are NameMatchTransactionAttributeSource. or MethodMapTransactionAttributeSource.

- MatchAlwaysTransactionAttributeSource is configured with a ITransactionAttribute instance that is applied to all methods. The shorthand string representation, i.e. PROPAGATION_REQUIRED can be used
- AttributesTransactionAttributeSource : Use a standard. .NET attributes to specify the transactional information. See TransactionAttribute class for more information.
- NameMatchTransactionAttributeSource allows ITransactionAttributes to be matched by method name. The NameMap IDictionary property is used to specify the mapping. For example

Key values can be prefixed and/or suffixed with wildcards as well as include the full namespace of the containing class.

• MethodMapTransactionAttributeSource : Similar

to NameMatchTransactionAttributeSource but specifies that only fully qualified method names (i.e. type.method, assembly) and wildcards can be used at the start or end of the method name for matching multiple methods.

• DefaultAdvisorAutoProxyCreator: looks for Advisors in the context, and automatically creates proxy objects which are the transactional wrappers

Refer to the following section for a more convenient way to achieve the same goal of declarative transaction management using attributes.

17.5.3. Declarative transactions using the transaction namespace

Spring provides a custom XML schema to simplify the configuration of declarative transaction management. If you would like to perform attribute driven transaction management you first need to register the custom namespace parser for the transaction namespace. This can be done in the application configuration file as shown below

```
</spring>
</configSections>
```

Instead of using the XML configuration listed at the end of the previous section (declarativeServices.xml you can use the following. Note that the schemaLocation in the objects element is needed only if you have not installed Spring's schema into the proper VS.NET 2005 location. See the chapter on VS.NET integration for more details.

```
<objects xmlns="http://www.springframework.net"</pre>
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-
         xmlns:tx="http://www.springframework.net/tx"
         xmlns:db="http://www.springframework.net/database"
         xsi:schemaLocation="http://www.springframework.net
         http://www.springframework.net/schema/tx http://www
         http://www.springframework.net/schema/db http://www
    <db:provider id="DbProvider"</pre>
                   provider="SqlServer-1.1"
                   connectionString="Data Source=(local);Data
    <object id="transactionManager"</pre>
            type="Spring.Data.AdoPlatformTransactionManager,
        property name="DbProvider" ref="DbProvider"/>
    </object>
    <object id="adoTemplate" type="Spring.Data.AdoTemplate,</pre>
        property name="DbProvider" ref="DbProvider"/>
    </object>
    <object id="testObjectDao" type="Spring.Data.TestObject[</pre>
        property name="AdoTemplate" ref="adoTemplate"/>
    </object>
    <!-- The object that performs multiple data access opera
    <object id="testObjectManager"</pre>
            type="Spring.Data.TestObjectManager, Spring.Data
        cproperty name="TestObjectDao" ref="testObjectDao"/>
```

</object>

<tx:attribute-driven transaction-manager="transactionMar

</objects>



Tip

You can actually omit the 'transaction-manager' attribute in the <tx:attribute-driven/> tag if the object name of the IPlatformTransactionManager that you want to wire in has the name 'transactionManager'. If the PlatformTransactionManager object that you want to dependency inject has any other name, then you have to be explicit and use the 'transaction-manager' attribute as in the example above.

The various optional elements of the <tx:attribute-driven/> tag are summarised in the following table

Table 17.1. <tx:annotation-driven/> settings

Attribute Required? **Default Description**

transactionmanager

The name of transaction manager to use. Only required if the name of the transactionManager transaction manager is not transactionManager, as in the example above.

> Controls what type of transactional proxies are created for classes annotated with the [Transaction] attribute. If "proxy-target-type" attribute is set to "true", then class-based proxies will be created (proxy inherits from target class, however calls are still delegated to target object via composition. This allows for casting to base class. If "proxytarget-type" is "false" or if the attribute is omitted, then a pure composition based proxy is created and you can only cast the proxy to

proxy-No target-type

implemented interfaces. (See the section entitled <u>Section 13.6</u>, "<u>Proxying mechanisms</u>" for a detailed examination of the different proxy types.)

Defines the order of the transaction advice that will be applied to objects annotated with [Transaction]. More on the rules related to ordering of AOP advice can be found in the AOP chapter (see section Section 13.3.2.5, "Advice Ordering"). Note that not specifying any ordering will leave the decision as to what order advice is run in to the AOP subsystem.

order No



Note

The "proxy-target-type" attribute on the <tx:attribute-driven/> element controls what type of transactional proxies are created for classes annotated with the Transaction attribute. If "proxy-target-type" attribute is set to "true", then inheritance-based proxies will be created. If "proxy-target-type" is "false" or if the attribute is omitted, then composition based proxies will be created. (See the section entitled Section 13.6, "Proxying mechanisms" for a detailed examination of the different proxy types.)

You can also define the transactional semantics you want to apply through the use of a <tx:advice> definition. This lets you define the transaction metadata such as propagation and isolation level as well as the methods for which that metadata applies external to the code unlike the case of using the transaction attribute. The <tx:advice> definition creates an instance of a ITransactionAttributeSource during parsing time. Switching to use <tx:advice> instead of <tx:attribute-driven/> in the example would look like the following

```
</tx:attributes>
</tx:advice>
```

This says that all methods that start with Save and Delete would have associated with them the default settings of transaction metadata. These default values are listed below..

Here is an example using other elements of the <tx:method/> definition

The <tx:advice/> definition reads as "... all methods on starting with 'Get' are to execute in the context of a read-only transaction, and all other methods are to execute with the default transaction semantics". The 'transaction-manager' attribute of the <tx:advice/> tag is set to the name of the PlatformTransactionManager object that is going to actually drive the transactions (in this case the 'transactionManager' object).

You can also use the AOP namespace <aop:advisor> element to tie together a pointcut and the above defined advice as shown below.

This is assuming that the service layer class, TestObjectManager, in the

namespace Spring.TxQuickStart.Services. The <aop:config/> definition ensures that the transactional advice defined by the 'txAdvice' object actually executes at the appropriate points in the program. First we define a pointcut that matches any operation defined on classes in the Spring.TxQuickStart.Services (you can be more selective in your regular expression). Then we associate the pointcut with the 'txAdvice' using an advisor. In the example, the result indicates that at the execution of a 'SaveTwoTestObjects' and 'DeleteTwoTestObject', the advice defined by 'txAdvice' will be run.

The various transactional settings that can be specified using the <tx:advice/> tag. The default <tx:advice/> settings are listed below and are the same as when you use the Transaction attribute.

- The propagation setting is TransactionPropagation.Required
- The isolation level is IsolationLevel.ReadCommitted
- The transaction is read/write
- The transaction timeout defaults to the default timeout of the underlying transaction system, or none if timeouts are not supported
- EnterpriseServicesInteropOption (.NET 2.0 only with TxScopeTransactionManager) options between transaction created with System.Transactions and transactions created through COM+
- Any exception will trigger rollback.

These default settings can be changed; the various attributes of the <tx:method/> tags that are nested within <tx:advice/> and <tx:attributes/> tags are summarized below:

Table 17.2. <tx:method></tx:method> settings				
	Attribute	Required?	Default	
name		Yes		The met associate transacti 'Get*
propagation	1	No	Required	The tra
		No	ReadCommitted	The tra

isolation			
timeout	No	-1	The tra
read-only	No	false	Is this t
EnterpriseServicesInteropOption	No	None	Interope TxScop
rollback-for	No		The Exc example 'MyPro
no-rollback-for	No		The Exc example 'MyPro

17.5.4. Transaction attribute settings

The Transaction attribute is metadata that specifies that a class or method must have transactional semantics. The default Transaction attribute settings are

- The propagation setting is TransactionPropagation.Required
- The isolation level is IsolationLevel.ReadCommitted
- The transaction is read/write
- The transaction timeout defaults to the default timeout of the underlying transaction system, or none if timeouts are not supported
- EnterpriseServicesInteropOption (.NET 2.0 only with TxScopeTransactionManager) options between transaction created with System.Transactions and transactions created through COM+
- Any exception will trigger rollback.

The default settings can, of course, be changed; the various properties of the Transaction attribute are summarised in the following table

Table 17.3. Transaction attribute properties

Property	Туре
TransactionPropagation	enumeration, Spring.Transaction.TransactionPropag
Isolation	System.Data.IsolationLevel
ReadOnly	boolean
EnterpriseServicesInteropOption	enumeration System.Transactions.EnterpriseServicesInteropOption
Timeout	int (in seconds granularity)
RollbackFor	an array of Type objects
NoRollbackFor	an array of Type objects

Note that setting the TransactionPropagation to Nested will throw a NestedTransactionNotSupportedException in a case where an actual nested transaction occurs, i.e. not in the case of applying the Nested propagation but in fact no nested calls are made. This will be fixed for the Spring 1.2 release for SqlServer and Oracle which support nested transactions. Also note, that changing of isolation levels on a per-method basis is also scheduled for the Spring 1.2 release since it requires detailed command text metadata for each dbprovider. Please check the forums for news on when this feature will be introduced into the nightly builds.

If you specify an exception type for 'NoRollbackFor' the action taken is to commit the work that has been done in the database up to the point where the exception occurred. The exception is still propagated out to the calling code.

The ReadOnly boolean is a hint to the data access technology to enable read-only optimizations. This currently has no effect in Spring's ADO.NET framework. If you would like to enable read-only optimizations in ADO.NET this is generally done via the 'Mode=Read' or 'Mode=Read-Only" options in the connection string. Check your database provider for more information. In the case of NHibernate the flush mode is set to Never when a new Session is created for the transaction.

Throwing exceptions to indicate failure and assuming success is an easier and less invasive programming model than performing the same task Programatically - ContextUtil.MyTransactionVote or TransactionScope.Complete. The rollback options are a means to influence the outcome of the transaction based on the exception type which adds an extra degree of flexibility.

Having any exception trigger a rollback has similar behavior as applying the AutoComplete attribute available when using .NET Enterprise Services. The difference with AutoComplete is that using AutoComplete is also coupled to the lifetime of the ServicedComponent since it sets ContextUtil.DeactivateOnReturn to true. For a stateless DAO layer this is not an issue but it could be in other scenarios. Spring's transactional aspect does not affect the lifetime of your object.

17.5.5. Declarative Transactions using AutoProxy

if you choose not to use the transaction namespace for declarative transaction management then you can use 'lower level' object definitions to configure declarative transactions. This approach was shown in the first example. The use of Spring's autoproxy functionality defines criteria to select a collection of objects to create a transactional AOP proxy. There are two AutoProxy classes that you can use, ObjectNameAutoProxyCreator and DefaultAdvisorAutoProxyCreator. If you are using the new transaction namespace support you do not need to configure these objects as a DefaultAdvisorAutoProxyCreator is created 'under the covers' while parsing the transaction namespace elements

17.5.5.1. Creating transactional proxies with ObjectNameAutoProxyCreator

The ObjectNameAutoProxyCreator is useful when you would like to create

transactional proxies for many objects. The definitions for the TransactionInterceptor and associated attributes is done once. When you add new objects to your configuration file that need to be proxies you only need to add them to the list of object referenced in the ObjectNameAutoProxyCreator. Here is an example showing its use. Look in the section that use ProxyFactoryObject for the declaration of the transactionInterceptor.

17.5.5.2. Creating transactional proxies with DefaultAdvisorAutoProxyCreator

This is a commonly used way to configure declarative transactions since it enables you to refer to the transaction attribute as the pointcut to use for the transactional advice for any object definition defined in the IoC container. An example of this configuration approach was shown in Chapter 5.

17.5.6. Declarative Transactions using TransactionProxyFactoryObject

The TransactionProxyFactoryObject is easier to use than a ProxyFactoryObject for most cases since the transaction interceptor and transaction attributes are properties of this object. This removes the need to declare them as separate objects. Also, unlike the case with the ProxyFactoryObject, you do not have to give fully qualified method names, just the normal 'short' method name. Wild card matching on the method name is also allowed, which in practice helps to enforce a common naming convention for the methods of your DAOs. The example from chapter 5 is shown here using a TransactionProxyFactoryObject.

```
<object id="testObjectManager"</pre>
```

Note the use of an inner object definition for the target which will make it impossible to obtain an unproxied reference to the TestObjectManager.

As can be seen in the above definition, the TransactionAttributes property holds a collection of name/value pairs. The key of each pair is a method or methods (a * wildcard ending is optional) to apply transactional semantics to. Note that the method name is not qualified with a package name, but rather is considered relative to the class of the target object being wrapped. The value portion of the name/value pair is the TransactionAttribute itself that needs to be applied. When specifying it as a string value as in this example, it's in String format as defined by TransactionAttributeConverter. This format is:

PROPAGATION_NAME, ISOLATION_NAME, readOnly, timeou Exception2

Note that the only mandatory portion of the string is the propagation setting. The default transactions semantics which apply are as follows:

- Exception Handling: All exceptions thrown trigger a rollback.
- Transactions are read/write
- Isolation Level: TransactionDefinition.ISOLATION_DEFAULT
- Timeout: TransactionDefinition.TIMEOUT_DEFAULT

Multiple rollback rules can be specified here, comma-separated. A - prefix

forces rollback; a + prefix specifies commit. Under the covers the IDictionary of name value pairs will be converted to an instance of NameMatchTransactionAttributeSource

The string used for PROPAGATION_NAME are those defined on the Spring.Transaction.TransactionPropagation enumeration, namely Required, Supports, Mandatory, RequiresNew, NotSupported, Never, Nested. The string used for ISOLATION_NAME are those defined on the System.Data.IsolationLevel enumberateion, namely ReadCommitted, ReadUncommitted, RepeatableRead, Serializable.

The TransactionProxyFactoryObject allows you to set optional "pre" and "post" advice, for additional interception behavior, using the "PreInterceptors" and "PostInterceptors" properties. Any number of pre and post advices can be set, and their type may be Advisor (in which case they can contain a pointcut), MethodInterceptor or any advice type supported by the current Spring configuration (such as ThrowsAdvice, AfterReturningAdvice or BeforeAdvice, which are supported by default.) These advices must support a shared-instance model. If you need transactional proxying with advanced AOP features such as stateful mixins, it's normally best to use the generic ProxyFactoryObject, rather than the TransactionProxyFactoryObject convenience proxy creator.

17.5.7. Concise proxy definitions

Using abstract object definitions in conjunction with a TransactionProxyFactoryObject provides you a more concise means to reuse common configuration information instead of duplicating it over and over again with a definition of a TransactionProxyFactoryObject per object. Objects that are to be proxied typically have the same pattern of method names, Save*, Find*, etc. This commonality can be placed in an abstract object definition, which other object definitions refer to and change only the configuration information that is different. An abstract object definition is shown below

Subsequent definitions can refer to this 'base' configuration as shown below

17.5.8. Declarative Transactions using ProxyFactoryObject

Using the general ProxyFactoryObject to declare transactions gives you a great deal of control over the proxy created since you can specify additional advice, such as for logging or performance. Based on the example shown previously a sample configuration using ProxyFactoryObject is shown below

The ProxyFactoryObject will create a proxy for the Target, i.e. a

TestObjectManager instance. An inner object definition could also have been used such that it would make it impossible to obtain an unproxied object from the container. The interceptor name refers to the following definition.

```
<object id="transactionInterceptor" type="Spring.Transaction"</pre>
   property name="TransactionManager" ref="adoTransact
   <!-- note do not have converter from string to this
   </object>
<object name="methodMapTransactionAttributeSource"</pre>
       type="Spring.Transaction.Interceptor.MethodMapTr
   property name="MethodMap">
      <dictionary>
         <entry key="Spring.Data.TestObjectManager.Save</pre>
                value="PROPAGATION_REQUIRED"/>
         <entry key="Spring.Data.TestObjectManager.Dele</pre>
                value="PROPAGATION REQUIRED"/>
      </dictionary>
   </property>
</object>
```

The transaction options for each method are specified using a dictionary containing the class name + method name, assembly as the key and the value is of the form

• <Propagation Behavior>, <Isolation Level>, <ReadOnly>, -Exception, +Exception

All but the propagation behavior are optional. The + and - are used in front of the name of an exception. Minus indicates to rollback if the exception is thrown, the Plus indicates to commit if the exception is thrown.

17.6. Programmatic transaction management

Spring provides two means of programmatic transaction management:

- Using the TransactionTemplate
- Using a IPlatformTransactionManager implementation directly

These are located in the Spring. Transaction. Support namespace. If you are going to use programmatic transaction management, the Spring team generally recommends the first approach (i.e. Using the TransactionTemplate)

17.6.1. Using the TransactionTemplate

The TransactionTemplate adopts the same approach as other Spring templates such as AdoTemplate and HibernateTemplate. It uses a callback approach, to free application code from having to do the boilerplate acquisition and release of resources, and results in code that is intention driven, in that the code that is written focuses solely on what the developer wants to do. Granted that the using construct of System. Transaction alleviates much of this. One key difference with the approach taken with the TransactionTemplate is that a commit is assumed - throwing an exception triggers a rollback instead of using the TransactionScope API to commit or rollback. This also allows for the use of rollback rules, that is a commit can still occur for exceptions of certain types.



As you will immediately see in the examples that follow, using the TransactionTemplate absolutely couples you to Spring's transaction infrastructure and APIs. Whether or not programmatic transaction management is suitable for your development needs is a decision that you will have to make yourself.

Application code that must execute in a transaction context looks like this. You, as an application developer, will write a ITransactionCallback implementation (typically expressed as an anonymous delegate) that will contain all of the code that you need to have execute in the context of a transaction. You will then pass an instance of your custom ITransactionCallback to the Execute(..) method on the TransactionTemplate. that the exposed ITransactionCallback can be used to return a value:

This code example is specific to .NET 2.0 since it uses anonymous delegates, which provides a particularly elegant means to invoke a callback function as local variables can be referred to inside the delegate, i.e. userId. In this case the ITransactionStatus was not exposed in the delegate (delegate can infer the signature to use), but one could also obtain a reference to the ITransactionStatus instance and set the RollbackOnly property to trigger a rollback - or alternatively throw an exception. This is shown below

```
tt.Execute(delegate(ITransactionStatus status)
{
    try {
        UpdateOperation1();
        UpdateOperation2();
    } catch (SomeBusinessException ex) {
        status.RollbackOnly = true;
    }
    return null;
});
```

If you are using .NET 1.1 then you should provide a normal delegate reference

or an instance of a class that implements the ITransactionCallback interface. This is shown below

Application classes wishing to use the TransactionTemplate must have access to a IPlatformTransactionManager (which will typically be supplied to the class via dependency injection). It is easy to unit test such classes with a mock or stub IPlatformTransactionManager.

17.6.1.1. Specifying transaction settings

Transaction settings such as the propagation mode, the isolation level, the timeout, and so forth can be set on the TransactionTemplate either programmatically or in configuration. TransactionTemplate instances by default have the default transactional settings. Find below an example of programmatically customizing the transactional settings for a specific TransactionTemplate.

```
public class SimpleService : IService
{
```

```
private TransactionTemplate transactionTemplate;

public SimpleService(IPlatformTransactionManager transacti
{
    AssertUtils.ArgumentNotNull(transactionManager, "transactionTemplate = new TransactionTemplate(transacti
    // the transaction settings can be set here explicitly
    transactionTemplate.TransactionIsolationLevel = Isolati
    transactionTemplate.TransactionTimeout = 30;

    // and so forth...
}
. . . .
```

Find below an example of defining a TransactionTemplate with some custom transactional settings, using Spring XML configuration. The 'sharedTransactionTemplate' can then be injected into as many services as are required.

Finally, instances of the TransactionTemplate class are threadsafe, in that instances do not maintain any conversational state. TransactionTemplate instances do however maintain configuration state, so while a number of classes may choose to share a single instance of a TransactionTemplate, if a class needed TransactionTemplate with different settings (for example, a different isolation level), then two distinct TransactionTemplate instances would need to be created and used.

17.6.2. Using the PlatformTransactionManager

You can also use the PlatformTransactionManager directly to manage your transaction. Simply pass the implementation of the PlatformTransactionManager you're using to your object via a object reference through standard Dependency Injection techniques. Then, using the TransactionDefinition and ITransactionStatus objects, you can initiate transactions, rollback and commit.

```
DefaultTransactionDefinition def = new DefaultTransactionDef
def.PropagationBehavior = TransactionPropagation.Required;

ITransactionStatus status = transactionManager.GetTransactic

try
{
    // execute your business logic here
} catch (Exception e)
{
    transactionManager.Rollback(status);
    throw;
}
transactionManager.Commit(status);
```

Note that a corresponding 'using TransactionManagerScope' class can be modeled to get similar API usage to System. Transactions TransactionScope.

17.7. Choosing between programmatic and declarative transaction management

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that require transactions only for certain update operations, you may not want to set up transactional proxies using Spring or any other technology. In this case, using the TransactionTemplate may be a good approach. On the other hand, if your application has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure in Spring.

17.8. Transaction lifecycle and status information

You can query the status of the current Spring managed transaction with the class TransactionSynchronizationManager. Typical application code should not need to rely on using this class but in some cases it is convenient to receive events around the lifecycle of the transaction, i.e. before committing, after committing.

TransactionSynchronizationManager provides a method to register a callback object that is informed on all significant stages in the transaction lifecycle. Note that you can register for lifecycle call back information for any of the transaction managers you use, be it NHibernate or local ADO.NET transactions.

The method to register a callback with the TransactionSynchronizationManager is

```
public static void RegisterSynchronization( ITransactionSync
```

Please refer to the SDK docs for information on other methods in this class.

The ITransactionSynchronization interface is

```
public interface ITransactionSynchronization
{

    // Typically used by Spring resource management code
    void Suspend();
    void Resume();

    // Transaction lifeycyle callback methods
    // Typically used by Spring resource management code but n
    void BeforeCommit( bool readOnly );
    void AfterCommit();
    void BeforeCompletion();
    void AfterCompletion( TransactionSynchronizationStatus state)
```

The TransactionSynchronizationStatus is an enum with the values Committed, Rolledback, and Unknown.

Chapter 18. DAO support

18.1. Introduction

Spring promotes the use of data access interfaces in your application architecture. These interfaces encapsulate the storage and retrieval of data and objects specific to your business domain without reference to a specific persistence API. Within a layered architecture, the service layer is typically responsible for coordinating responses to a particular business request and it delegates any persistence related activities to objects that implement these data access interfaces. These objects are commonly referred to as DAOs (Data Access Objects) and the architectural layer as a DAL (Data Access Layer).

The benefits of using DAOs in your application are increased portability across persistence technology and ease of testing. Testing is more easily facilitated because a mock or stub implementation of the data access interface can be easily created in a NUnit test so that service layer functionality can be tested without any dependency on the database. This is beneficial because tests that rely on the database are usually hard to set up and tear down and also are impractical for testing exceptional behavior.

The Data Access Object (DAO) support in Spring is aimed at making it easy to work with data access technologies like ADO.NET and NHibernate in a standardized way. Spring provides two central pieces of functionality to meet this goal. The first is providing a common exception hierarchy across providers and the second is providing base DAOs classes that raise the level of abstraction when performing common ADO.NET operations. This allows one to switch between the aforementioned persistence technologies fairly easily and it also allows one to code without worrying about catching exceptions that are specific to each technology.

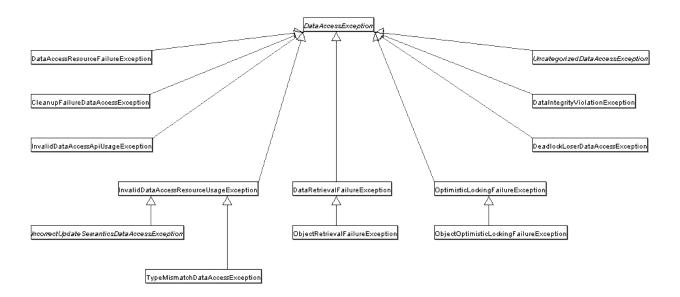
18.2. Consistent exception hierarchy

Database exceptions in the ADO.NET API are not consistent across providers. The .NET 1.1 BCL did not provide a common base class for ADO.NET exceptions. As such you were required to handle exceptions specific to each provider such as System.Data.SqlClient.SqlException or System.Data.OracleClient.OracleException. The .NET 2.0 BCL improved in this regard by introducing a common base class for exceptions, System.Data.Common.DbException. However the common DbException is not very portable either as it provides a vendor specific error code as the underlying piece of information as to what went wrong. This error code is different across providers for the same conceptual error, such as a violation of data integrity or providing bad SQL grammar.

To promote writing portable and descriptive exception handling code Spring provides a convenient translation from technology specific exceptions like System.Data.SqlClient.SqlException or System.Data.OracleClient.OracleException to its own exception hierarchy with the Spring.Dao.DataAccessException as the root exception. These exceptions wrap the original exception so there is never any risk that one might lose any information as to what might have gone wrong.

In addition to exceptions from ADO.NET providers, Spring can also wrap NHibernate-specific exceptions.. This allows one to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without boilerplate using or catch and throw blocks, and exception declarations. As mentioned above, ADO.NET exceptions (including database-specific dialects) are also converted to the same hierarchy, meaning that one can perform some operations with ADO.NET within a consistent programming model. The above holds true for the various template-based versions of the ORM access framework.

The exception hierarchy that Spring uses is outlined in the following image:



(Please note that the class hierarchy detailed in the above image shows only a subset of the whole, rich, DataAccessException hierarchy.)

The exception translation functionality is in the namespace Spring.Data.Support and is based on the interface IAdoExceptionTranslator shown below.

```
public interface IAdoExceptionTranslator
{
   DataAccessException Translate( string task, string sql, Ex
}
```

The arguments to the translator are a task string providing a description of the task being attempted, the SQL query or update that caused the problem, and the 'raw' exception thrown by the ADO.NET data provider. The additional task and SQL arguments allow for very readable and clear error messages to be created when an exception occurs.

A default implementation, ErrorCodeExceptionTranslator, is provided that uses the error codes defined for each data provider in the file dbproviders.xml. Refer to this file, an embedded resource in the Spring.Data assembly, for the exact mappings of error codes to Spring DataAccessExceptions.

A common need is to modify the error codes that are map onto the exception

hierarchy. There are several ways to accomplish this task.

One approach is to override the error codes that are defined in assembly://Spring.Data/Spring.Data.Common/dbpro By default, the DbProviderFactory will look for additional metadata for the IoC container it uses internally to define and manage the DbProviders in a file named dbProviders.xml located in the root runtime directory. (You can change this location, see the documentation on DbProvider for more information.) This is Spring application context SO all standard features. ObjectFactoryPostProcessors are available and will be automatically applied. Defining a PropertyOverrideConfigurer in this additional configuration file will allow for you to override specific property values defined in the embedded resource file. As an example, the additional dbProviders.xml file shown below will add the error code 2601 to the list of error codes that map to a DataIntegrityViolationException.

The reason to define the alias is that PropertyOverrideConfigurer assumes a period (.) as the separator to pick out the object name but the names of the objects in dbProviders.xml have periods in them (i.e. SqlServer-2.0 or System.Data.SqlClient). Creating an alias that has no periods in the name is a workaround.

Another way to customize the mappings of error codes to exceptions is to subclass <code>ErrorCodeExceptionTranslator</code> and override the

method, DataAccessException

TranslateException(string task, string sql, string errorCode, Exception exception). This will be called before referencing the metadata to perform exception translation. The vendor specific error code provided as a method argument has already been parsed out of the raw ADO.NET exception. If you create your own specific subclass, then you should set the property ExceptionTranslator on AdoTemplate

HibernateTemplate/HibernateTransactionManager to refer to your custom implementation (unless you are using autowiring).

The third way is to write an implementation of IAdoExceptionTranslator and set the property FallbackTranslator'on

ErrorCodeExceptionTranslator. In this case you are responsible for parsing our the vendor specific error code from the raw ADO.NET exception. As with the case of subclassing ErrorCodeExceptionTranslator, you will need to refer to this custom exception translator when using AdoTemplate or

HibernateTemplate/HibernateTransactionManager.

The ordering of the exception translation processing is as follows. The method TranslateException is called first, then the standard exception translation logic, then the FallbackTranslator.

Note that you can use this API directly in your own Spring independent data layer. If you are using Spring's ADO.NET abstraction class, AdoTemplate, or HibernateTemplate, the converted exceptions will be thrown automatically. Somewhere in between these two cases is using Spring's declarative transaction management features in .NET 2.0 with the raw ADO.NET APIs and using IAdoExceptionTranslator in your exception handling layer (which might be implemented in AOP using Spring's exception translation aspect).

Some of the more common data access exceptions are described here. Please refer to the API documentation for more details.

 Table 18.1. Common DataAccessExceptions

Exception	Description
BadSqlGrammarException	Exception thrown when SQL specified is invalid.
DataIntegrityViolationException	Exception thrown when an attempt to insert or update data results in violation of an integrity constraint. For example, inserting a duplicate key.
PermissionDeniedDataAccessException	Exception thrown when the underling resource denied a permission to access a specific element, such as a specific database table.
DataAccessResourceFailureException	Exception thrown when a resource fails completely, for example, if we can't connect to a database.
ConcurrentyFailureException	Exception thrown when a concurrency error occurs. OptimisticLockingFailureException and PessimisticLockingFailureException are subclasses. This is a useful exception to catch and to retry the transaction again. See Spring's Retry Aspect for an AOP based solution.
OptimisticLockingFailureException	Exception thrown when there an optimistic locking failure occurs. The subclass ObjectOptimisticLockingFailureException can be used to examine the Type and the IDof the object that failed the optimistic locking.
PessimisticLockingFailure	Exception thrown when a pessimistic locking failure occures. Subclasses of this exception are CannotAcquireLockException, CannotSerializeTransactionException, and DeadlockLoserDataAccessException.
CannotAcquireLockException	Exception thrown when a lock can not be acquired, for example during an update, ie a select for update
CannotSerializeTransactionException	Exception thrown when a transaction can not be serialized.

18.3. Consistent abstract classes for DAO support

To make it easier to work with a variety of data access technologies such as ADO.NET, NHibernate, and iBatis.NET in a consistent way, Spring provides a set of abstract DAO classes that one can extend. These abstract classes have methods for providing the data source and any other configuration settings that are specific to the technology one is currently using.

DAO support classes:

- AdoDaoSupport super class for ADO.NET data access objects.
 Requires a DbProvider to be provided; in turn, this class provides a AdoTemplate instance initialized from the supplied DbProvider to subclasses. See the documentation for AdoTemplate for more information.
- HibernateDaoSupport super class for NHibernate data access objects. Requires a ISessionFactory to be provided; in turn, this class provides a HibernateTemplate instance initialized from the supplied SessionFactory to subclasses. Can alternatively be initialized directly via a HibernateTemplate, to reuse the latter's settings like SessionFactory, flush mode, exception translator, etc. This is contained in a download separate from the main Spring.NET distribution.

Chapter 19. DbProvider

19.1. Introduction

Spring provides a generic factory for creating ADO.NET API artifacts such as IDbConnection and IDbCommand. The factory API is very similar to the one introduced in .NET 2.0 but adds extra metadata needed by Spring to support features provided by its DAO/ADO.NET framework such as error code translation to a DAO exception hierarchy. The factory itself is configured by using a standard Spring XML based configuration file though it is unlikely you will need to modify those settings yourself, you only need be concerned with using the factory. Out of the box several popular databases are supported and an extension mechanism is available for defining new database providers or modifying existing ones. A custom database namespace for configuration aids in making terse XML based declarations of Spring's database objects you wish to use.

The downside of Spring's factory as compared to the one in .NET 2.0 is that the types returned are lower level interfaces and not the abstract base classes in System.Data.Common. However, there are still 'holes' in the current .NET 2.0 provider classes that are 'plugged' with Spring's provider implementation. One of the most prominent is the that the top level DbException exposes the HRESULT of the remote procedure call, which is not what you are commonly looking for when things go wrong. As such Spring's provider factory exposes the vendor sql error code and also maps that error code onto a consistent data access exception hierarchy. This makes writing portable exception handlers much easier. In addition, the DbParameter class doesn't provide the most common convenient methods you would expect as when using say the SqlServer provider. If you need to access the BCL provider abstraction, you still can through Spring's provider class. Furthermore, a small wrapper around the standard BCL provider abstraction allows for integration with Spring's transaction management facilities, allowing you to create a DbCommand with its connection and transaction properties already set based on the transaction calling context.

19.2. IDbProvider and DbProviderFactory

The **IDbProvider** API is shown below and should look familiar to anyone using .NET 2.0 data providers. Note that Spring's DbProvider abstraction can be used on .NET 1.1 in addition to .NET 2.0

```
public interface IDbProvider
{
  IDbCommand CreateCommand();
  object CreateCommandBuilder();
  IDbConnection CreateConnection();
  IDbDataAdapter CreateDataAdapter();
  IDbDataParameter CreateParameter();
  string CreateParameterName(string name);
  string CreateParameterNameForCollection(string name);
  IDbMetadata DbMetadata
  {
      get;
  }
  string ConnectionString
      set;
      get;
  }
  string ExtractError(Exception e);
  bool IsDataAccessException(Exception e);
}
```

ExtractError is used to return an error string for translation into a DAO exception. On .NET 1.1 the method IsDataAccessException is used to determine

if the thrown exception is related to data access since in .NET 1.1 there isn't a common base class for database exceptions. CreateParameterName is used to create the string for parameters used in a CommandText object while CreateParameterNameForCollection is used to create the string for a IDataParameter.ParameterName, typically contained inside a IDataParameterCollection.

The class <code>DbProviderFactory</code> creates <code>IDbProvider</code> instances given a provider name. The connection string property will be used to set the <code>IDbConnection</code> returned by the factory if present. The provider names, and corresponding database, currently configured are listed below.

- SqlServer-1.1 Microsoft SQL Server, provider V1.0.5000.0 in framework .NET V1.1
- SqlServer-2.0 (aliased to System.Data.SqlClient) Microsoft SQL Server, provider V2.0.0.0 in framework .NET V2.0
- SqlServerCe-3.1 Microsoft SQL Server Compact Edition, provider V9.0.242.0
- SqlServerCe-3.5.1 (aliased to System.Data.SqlServerCe) Microsoft SQL Server Compact Edition, provider V3.5.1.0
- OleDb 1 . 1 OleDb, provider V1.0.5000.0 in framework .NET V1.1
- OleDb-2.0 (aliased to System.Data.OleDb) OleDb, provider V2.0.0.0 in framework .NET V2.0
- OracleClient-2.0 (aliased to System.Data.OracleClient) Oracle, Microsoft provider V2.0.0.0
- OracleODP-2.0 (aliased to System.DataAccess.Client)
 Oracle, Oracle provider V2.102.2.20
- MySql MySQL, MySQL provider 1.0.10.1
- MySql-1.0.9 MySQL, MySQL provider 1.0.9
- MySql-5.0 MySQL, MySQL provider 5.0.7.0
- MySql-5.0.8.1 MySQL, MySQL provider 5.0.8.1

- MySql-5.1 MySQL, MySQL provider 5.1.2.2
- MySql-5.1.4 MySQL, MySQL provider 5.1.2.2
- MySql-5.2.3 (aliased to MySql.Data.MySqlClient) MySQL, MySQL provider 5.2.3.0
- Npgsql-1.0 Postgresql provider 1.0.0.0 (and 1.0.0.1 were build with same version info)
- Npgsql-2.0-beta1 Postgresql provider 1.98.1.0 beta 1
- Npgsql-2.0 Postgresql provider 2.0.0.0
- DB2-9.0.0-1.1 IBM DB2 Data Provider 9.0.0 for .NET Framework 1.1
- DB2-9.0.0-2.0 (aliased to IBM.Data.DB2) IBM DB2 Data Provider 9.0.0 for .NET Framework 2.0
- DB2-9.1.0-1.1 IBM DB2 Data Provider 9.1.0 for .NET Framework 1.1
- DB2-9.1.0.2 (aliased to IBM.Data.DB2.9.1.0) IBM DB2 Data Provider 9.1.0 for .NET Framework 2.0
- **SQLite-1.0.43** SQLite provider 1.0.43 for .NET Framework 2.0
- SQLite-1.0.47 (aliased to System.Data.SQLite) SQLite provider 1.0.43 for .NET Framework 2.0
- SybaseAse 12 Sybase ASE provider for ASE 12.x
- SybaseAse 15 Sybase ASE provider for ASE 15.x
- SybaseAse-AdoNet2 Sybase ADO.NET 2.0 provider for ASE 12.x and 15.x
- Odbc-1.1 ODBC provider V1.0.5000.0 in framework .NET V1.1
- Odbc 2.0 ODBC provider V2.0.0.0 in framework .NET V2



If your exact version of the database provider is not listed, you can pick the gene provider name, i.e. MySql.Data.MySqlClient, and then perform an as redirect in App.config. This will often be sufficient to upgrade to newer versions shown below

An example using DbProviderFactory is shown below

```
IDbProvider dbProvider = DbProviderFactory.GetDbProvider("Sy
```

The default definitions of the providers are contained in the assembly resource assembly://Spring.Data/Spring.Data.Common/dbpro Future additions to round out the database coverage are forthcoming. The current crude mechanism to add additional providers, or to apply any standard Spring **IApplicationContext** functionality, such as applying AOP advice, is public static set the property DBPROVIDER ADDITIONAL RESOURCE NAME **DbProviderFactory** to a Spring resource location. The default value is file://dbProviders.xml. (That isn't a typo, there is a difference in case with the name of the embedded resource). This crude mechanism will eventually be replaced with one based on a custom configuration section in App.config/Web.config.

It may happen that the version number of an assembly you have downloaded is different than the one listed above. If it is a point release, i.e. the API hasn't changed in anyway that is material to your application, you should add an assembly redirect of the form shown below.

This redirects any reference to an older version of the assembly MySql.Data to the version 1.0.10.1.

19.3. XML based configuration

Creating a DbProvider in Spring's XML configuration file is shown below in the typical case of using it to specify the DbProvider property on an AdoTemplate.

A custom namespace should be registered in the main application configuration file to use this syntax. This configuration, only for the parsers, is shown below. Additional section handlers are needed to specify the rest of the Spring configuration locations as described in previous chapters.

19.4. Connection String management

There are a few options available to help manage your connection strings.

The first option is to leverage the Spring property replacement functionality, as described in <u>Section 5.9.2.1</u>, "<u>Example: The PropertyPlaceholderConfigurer</u>". This lets you insert variable names as placeholders for values in a Spring configuration file. In the following example specific parts of a connection string have been parameterized but you can also use a variable to set the entire connection string.

An example of such a setting is shown below

```
<configuration>
        <configSections>
                 <sectionGroup name="spring">
                          <section name='context' type='Spring.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Context.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Support.Supp
                 </sectionGroup>
                 <section name="databaseSettings" type="System.Configurat</pre>
        </configSections>
        <spring>
                 <context>
                          <resource uri="Aspects.xml" />
                          <resource uri="Services.xml" />
                          <resource uri="Dao.xml" />
                 </context>
        </spring>
        <!-- These properties are referenced in Dao.xml -->
        <databaseSettings>
                 <add key="db.datasource" value="(local)" />
                 <add key="db.user" value="springga" />
                 <add key="db.password" value="springga" />
                 <add key="db.database" value="Northwind" />
        </databaseSettings>
</configuration>
```

Where Dao.xml has a connection string as shown below

Please refer to the Section <u>Section 5.9.2.1</u>, "Example: The <u>PropertyPlaceholderConfigurer</u>" for more information.

19.5. Additional IDbProvider implementations

Spring provides some convenient implementations of the IDbProvider interface that add additional behavior on top of the standard implementation.

19.5.1. UserCredentialsDbProvider

This UserCredentialsDbProvider will allow you to change the username and password of a database connection at runtime. The API contains the properties Username and Password which are used as the default strings representing the user and password in the connection string. You can then change the value of these properties in the connection string by calling the method SetCredentialsForCurrentThread and fall back to the default values calling method by the RemoveCredentialsFromCurrentThread. call You SetCredentialsForCurrentThread method at runtime, before any data access occurs, to determine which database user should be used for the current user-case. Which user to select is up to you. You may retrieve the user information from an HTTP session for example. Example configuration and usage is shown below

If you use dependency injection to configure a class with a property of the type IDbProvider, you will need to downcast to the subtype or you can change your class to have a property of the type UserCredentialsDbProvider instead of IDbProvider.

```
userCredentialsDbProvider.SetCredentialsForCurrentThread("Us
```

DelegatingDbProvider, and is intended for you to use in your own implementations that delegate calls to a target IDbProvider instance. This class in meant to be subclassed with subclasses overriding only those methods, such as CreateConnection(), that should not simply delegate to the target IDbProvider.

19.5.2. MultiDelegatingDbProvider

There are use-cases in which there will need to be a runtime selection of the database to connect to among many possible candidates. This is often the case where the same schema is installed in separate databases for different clients. The MultiDelegatingDbProvider implements the IDbProvider interface and provides an abstraction to the multiple databases and can be used in DAO layer such that the DAO layer is unaware of the switching between databases. MultiDelegatingDbProvider does its job by looking into thread local storage under the key dbProviderName. This storage location stores the name of the dbProvider that is to be used for processing the request. MultiDelegatingDbProvider is configured using the dictionary property TargetDbProviders. The key of this dictionary contains the name of a dbProvider and its value is a dbProvider object. (You can also provide this dictionary as a constructor argument.)

During request processing, once you have determined which target dbProvider should be use, in this example database1ProviderName, you should execute the following code is you are using Spring 1.2 M1 or later

```
// Spring 1.2 M1 or later
LogicalThreadContext.SetData(MultiDelegatingDbProvider.CURRE
```

and the following ocde if you are using earlier versions

```
// Prior to Spring 1.2 M1
LogicalThreadContext.SetData("dbProviderName", "database1Pro
```

and then call the data access layer.

Here is a sample configuration to build up an object definition for MultiDelegatingDbProvider.

```
<db:provider id="CreditAndDebitsDbProvider"
    provider="System.Data.SqlClient"
    connectionString="Data Source=MARKT60\SQL2005;Initial Ca"

<db:provider id="CreditDbProvider"
    provider="System.Data.SqlClient"
    connectionString="Data Source=MARKT60\SQL2005;Initial Ca"

<object id="dbProviderDictionary" type="Spring.Collections.S"
    <property name="['DbProvider1']" ref="CreditAndDebitsDbProvider1']" ref="CreditAndDebitsDbProvider2']" ref="CreditDbProvider"/>
    </object>

<object id="DbProvider" type="Spring.Data.MultiDelegatingDbForoperty name="TargetDbProviders" ref="dbProviderDictiona"/object>
```

Chapter 20. Data access using ADO.NET

20.1. Introduction

Spring provides an abstraction for data access via ADO.NET that provides the following benefits and features

- Consistent and comprehensive database provider interfaces for both .NET 1.1 and 2.0
- Integration with Spring's transaction management features.
- Template style use of DbCommand that removes the need to write typical ADO.NET boiler-plate code.
- 'One-liner' implementations for the most common database usage patterns lets you focus on the 'meat' of your ADO.NET code.
- Easy database parameter creation/management
- Provider independent exceptions with database error codes and higher level DAO exception hierarchy.
- Centralized resource management for connections, commands, data readers, etc.
- Simple DataReader to Object mapping framework.

This chapter is divided up into a number of sections that describe the major areas of functionality within Spring's ADO.NET support.

- Motivations describes why one should consider using Spring's ADO.NET features as compared to using 'raw' ADO.NET API.
- Provider Abstraction a quick overview of Spring's provider abstraction.
- Approaches to ADO.NET Data Access Discusses the two styles of Spring's ADO.NET data access classes template and object based.
- Introduction to AdoTemplate Introduction to the design and core methods of the central class in Spring's ADO.NET support.
- Exception Translation Describes the features of Spring's data access exceptions
- Parameter Management Convenience classes and methods for easy parameter management.
- Custom IDataReader implementations Strategy for providing custom

implementations of IDataReader. This can be used to centralized and transparently map DBNull values to CLR types when accessing an IDataReader or to provide extended mapping functionality in sub-interfaces.

- Basic data access operations Usage of AdoTemplate for IDbCommand 'ExecuteScalar' and 'ExecuteNonScalar' functionality
- Queries and Lightweight Object Mapping Using AdoTemplate to map result sets into objects
- DataSet and DataTable operations Using AdoTemplate with DataSets and DataTables
- Modeling ADO.NET operations as .NET objects An object-oriented approach to data access operations.

. Motivations

There are a variety of motivations to create a higher level ADO.NET persistence API.

Encapsulation of common 'boiler plate' tasks when coding directly against the ADO.NET API. For example here is a list of the tasks typically required to be coded for processing a result set query. Note that the code needed when using Spring's ADO.NET framework is in italics.

- 1. Define connection parameters
- 2. Open the connection
- 3. *Specify the command type and text*
- 4. Prepare and execute the statement
- 5. Set up the loop to iterate through the results (if any)
- 6. Do the work for each iteration
- 7. Process any exception
- 8. Display or rollback on warnings
- 9. Handle transactions
- 10. Close the connection

Spring takes care of the low-level tasks and lets you focus on specifying the SQL and doing the real work of extracting data. This standard boiler plate pattern is encapsulated in a class, AdoTemplate. The name 'Template' is used because if you look at the typical code workflow for the above listing, you would essentially like to 'template' it, that is stick in the code that is doing the real work in the midst of the resource, transaction, exception management.

Another very important motivation is to provide an easy means to group multiple ADO.NET operations within a single transaction while at the same time adhering to a DAO style design in which transactions are initiated outside the DAOs, typically in a business service layer. Using the 'raw' ADO.NET API to implement this design often results in explicitly passing around of a Transaction/Connection pair to DAO objects. This infrastructure task distracts from the main database task at hand and is frequently done in an ad-hoc manner. Integrating with Spring's transaction management features provides an elegant

means to achieve this common design goal. There are many other benefits to integration with Spring's transaction management features, see Chapter 17. Transaction management for more information.

Provider Independent Code: In .NET 1.1 writing provider independent code was difficult for a variety of reasons. The most prominent was the lack of a lack of a central factory for creating interface based references to the core ADO.NET classes such as IDbConnection, IDbCommand, DbParameter etc. In addition, the APIs exposed by many of these interfaces were minimal or incomplete - making for tedious code that would otherwise be more easily developed with provider specific subclasses. Lastly, there was no common base class for data access exceptions across the providers. .NET 2.0 made many changes for the better in that regard across all these areas of concern - and Spring only plugs smaller holes in that regard to help in the portability of your data access code.

Resource Management: The 'using' block is the heart of elegant resource management in .NET from the API perspective. However, despite its elegance, writing 2-3 nested using statements for each data access method also starts to be tedious, which introduces the risk of forgetting to do the right thing *all the time* in terms of both direct coding and 'cut-n-paste' errors. Spring centralizes this resource management in one spot so you never forget or make a mistake and rely on it always being done correctly.

Parameter management: Frequently much of data access code is related to creating appropriate parameters. To alleviate this boiler plate code Spring provides a parameter 'builder' class that allows for succinct creation of parameter collections. Also, for the case of stored procedures, parameters can be derived from the database itself which reduces parameter creation code to just one line.

Frequently result set data is converted into objects. Spring provides a simple framework to organize that mapping task and allows you to reuse mapping artifacts across your application.

Exceptions: The standard course of action when an exception is thrown from ADO.NET code is to look up the error code and then re-run the application to set a break point where the exception occurred so as to see what the command text and data values were that caused the exception. Spring provides exceptions translation from these error codes (across database vendors) to a Data Access Object exception hierarchy. This allows you to quickly understand the category of the error that occurred and also the 'bad' data which lead to the exception.

Warnings: A common means to extract warning from the database, and to

optionally treat those warnings as a reason to rollback is not directly supported with the new System.Data.Common API

Portability: Where possible, increase the portability of code across database provider in the higher level API. The need adding of a parameter prefix, i.e. @ for SqlServer or ':' for oracle is one such example of an area where a higher level API can offer some help in making your code more portable.

Note that Spring's ADO.NET framework is just 'slightly' above the raw API. It does not try to compete with other higher level persistence abstractions such as result set mappers (iBATIS.NET) or other ORM tools (NHibernate). (Apologies if your favorite is left out of that short list). As always, pick and choose the appropriate level of abstraction for the task at hand. As a side note, Spring does offer integration with higher level persistence abstractions (currently NHibernate) providing such features as integration with Spring's transaction management features as well as mixing orm/ado.net operations within the same transaction.

20.3. Provider Abstraction

Before you get started executing queries against the database you need to connect to it. Chapter 19. DbProvider covers this topic in detail so we only discuss the basic idea of how to interact with the database in this section. One important ingredient that increases the portability of writing ADO.NET applications is to refer to the base ADO.NET interfaces, such as IDbCommand or IDbParameter in your code. However, In the .NET 1.1 BCL the only means to obtain references to instances of these interfaces is to directly instantiate the classes, i.e. for SqlServer this would be

```
IDbCommand command = new SqlCommand();
```

One of the classic creational patterns in the GoF Design Patterns book addresses this situation directly, the Abstract Factory pattern. This approach was applied in the .NET BCL with the introduction of the DbProviderFactory class which contains various factory methods that create the various objects used in ADO.NET programming. In addition, .NET 2.0 introduced new abstract base classes that all ADO.NET providers must inherit from. These base classes provide more core functionality and uniformity across the various providers as compared to the original ADO.NET interfaces.

Spring's database provider abstraction has a similar API to that of .ADO.NET 2.0's DbProviderFactory. The central interface is IDbProvider and it has factory methods that are analogous to those in the DbProviderFactory class except that they return references to the base ADO.NET interfaces. Note that in keeping with the Spring Framework's philosophy, IDbProvider is an interface, and can thus be easily mocked or stubbed as necessary. Another key element of this interface is the ConnectionString property that specifies the specific runtime information necessary to connect to the provider. The interface also has a IDbMetadata property that contains minimal database metadata information needed to support the functionality in rest of the Spring ADO.NET framework. It is unlikely you will need to use the DatabaseMetadata class directly in your application.

For more information on configuring a Spring database provider refer to Chapter 19, DbProvider

20.3.1. Creating an instance of IDbProvider

Each database vendor is associated with a particular implementation of the IDbProvider interfaces. A variety of implementations are provided with Spring such as SqlServer, Oracle and MySql. Refer to the documentation on Spring's DbProvider for creating a configuration for database that is not yet provided. The programmatic way to create an IDbProvider is shown below

IDbProvider dbProvider = DbProviderFactory.GetDbProvider("Sy

Please refer to the <u>Chapter 19</u>, <u>DbProvider</u> for information on how to create a IDbProvider in Spring's XML configuration file.

20.4. Namespaces

The ADO.NET framework consists of a few namespaces, namely Spring.Data, Spring.Data.Generic, Spring.Data.Common, Spring.Data.Support, and Spring.Data.Object.

The Spring.Data namespace contains the majority of the classes and interfaces you will deal with on a day to day basis.

The Spring.Data.Generic namespaces add generic versions of some classes and interfaces and you will also likely deal with this on a day to day basis if you are using .NET 2.0

The Spring.Data.Common namespaces contains Spring's DbProvider abstraction in addition to utility classes for parameter creation.

The Spring.Data.Object namespaces contains classes that represent RDBMS queries, updates, and stored procedures as thread safe, reusable objects.

Finally the Spring.Data.Support namespace is where you find the IAdoExceptionTransactor translation functionality and some utility classes.

20.5. Approaches to Data Access

Spring provides two styles to interact with ADO.NET. The first is a 'template' based approach in which you create an single instance of AdoTemplate to be used by all your DAO implementations. Your DAO methods are frequently implemented as a single method call on the template class as described in detail in the following section. The other approach a more object-oriented manner that models database operations as objects. For example, one can encapsulate the functionality of a database query via an AdoQuery class and a create/update/delete operation as a AdoNonQuery class. Stored procedures are also modelled in this manner via the class <code>StoredProcedure</code>. To use these classes you inherit from them and define the details of the operation in the constructor and implement an abstract method. This reads very cleanly when looking at DAO method implementation as you can generally see all the details of what is going on.

Generally speaking, experience has shown that the AdoTemplate approach reads very cleanly when looking at DAO method implementation as you can generally see all the details of what is going on as compared to the object based approach. The object based approach however, offers some advantages when calling stored procedures since it acts as a cache of derived stored procedure arguments and can be invoked passing a variable length argument list to the 'execute' method. As always, take a look at both approaches and use the approach that provides you with the most benefit for a particular situation.

20.6. Introduction to AdoTemplate

The class AdoTemplate is at the heart of Spring's ADO.NET support. It is based on an Inversion of Control (i.e. callback) design with the central method 'Execute' handing you a IDbCommand instance that has its Connection and Transaction properties set based on the transaction context of the calling code. All resource management is handled by the framework, you only need to focus on dealing with the IDbCommand object. The other methods in this class build upon this central 'Execute' method to provide you a quick means to execute common data access scenarios.

There are two implementations of AdoTemplate. The one that uses Generics and is in the namespace Spring.Data.Generic and the other non-generic version in Spring.Data. In either case you create an instance of an AdoTemplate by passing it a IDbProvider instance as shown below

```
AdoTemplate adoTemplate = new AdoTemplate(dbProvider);
```

AdoTemplate is a thread-safe class and as such a single instance can be used for all data access operations in you applications DAOs. AdoTemplate implements an IAdoOperations interface. Although the IAdoOperations interface is more commonly used for testing scenarios you may prefer to code against it instead of the direct class instance.

If you are using the generic version of AdoTemplate you can access the non-generic version via the property ClassicAdoTemplate.

The following two sections show basic usage of the AdoTemplate 'Execute' API for .NET 1.1 and 2.0.

20.6.1. Execute Callback

The Execute method and its associated callback function/inteface is the basic method upon which all the other methods in AdoTemplate delegate their work. If you can not find a suitable 'one-liner' method in AdoTemplate for your purpose you can always fall back to the Execute method to perform any database operation while benefiting from ADO.NET resource management and transaction enlistment. This is commonly the case when you are using special

provider specific features, such as XML or BLOB support.

20.6.2. Execute Callback in .NET 2.0

In this example a simple query against the 'Northwind' database is done to determine the number of customers who have a particular postal code.

The DbCommand that is passed into the anonymous delegate is already has it Connection property set to the corresponding value of the dbProvider instance used to create the template. Furthermore, the Transaction property of the DbCommand is set based on the transactional calling context of the code as based on the use of Spring's transaction management features. Also note the feature of anonymous delegates to access the variable 'postalCode' which is defined 'outside' the anonymous delegate implementation. The use of anonymous delegates is a powerful approach since it allows you to write compact data access code. If you find that your callback implementation is getting very long, it may improve code clarity to use an interface based version of the callback function, i.e. an ICommandCallback shown below.

As you can see, only the most relevant portions of the data access task at hand need to be coded. (Note that in this simple example you would be better off using AdoTemplate's ExecuteScalar method directly. This method is described in

the following sections). As mentioned before, the typical usage scenario for the Execute callback would involve downcasting the passed in **DbCommand** object to access specific provider API features.

There is also an interface based version of the execute method. The signatures for the delegate and interface are shown below

```
public delegate T CommandDelegate<T>(DbCommand command);

public interface ICommandCallback
{
   T DoInCommand<T>(DbCommand command);
}
```

While the delegate version offers the most compact syntax, the interface version allows for reuse. The corresponding method signatures on Spring.Data.Generic.AdoTemplate are shown below

While it is common for .NET 2.0 ADO.NET provider implementations to inherit from the base class System.Data.Common.DbCommand, that is not a requirement. To accommodate the few that don't, which as of this writing are the latest Oracle (ODP) provider, Postgres, and DB2 for iSeries, two additional execute methods are provided. The only difference is the use of callback and delegate implementations that have IDbCommand and not DbCommand as callback arguments. The following listing shows these methods on AdoTemplate.

```
public class AdoTemplate : AdoAccessor, IAdoOperations
```

```
{
    ...
    T Execute<T>(IDbCommandCallback action);
    T Execute<T>(IDbCommandDelegate<T> del);
    ...
}
```

where the signatures for the delegate and interface are shown below

```
public delegate T IDbCommandDelegate<T>(IDbCommand command);
public interface IDbCommandCallback<T>
{
    T DoInCommand(IDbCommand command);
}
```

Internally the AdoTemplate implementation delegates to implementations of IDbCommandCallback so that the 'lowest common denominator' API is used to have maximum portability. If you accidentally call Execute<T> (ICommandCallback action) and the command does not inherit from DbCommand, an

InvalidDataAccessApiUsageException will be thrown.

Depending on how portable you would like your code to be, you can choose among the two callback styles. The one based on <code>DbCommand</code> has the advantage of access to the more user friendly <code>DbParameter</code> class as compared to <code>IDbParameter</code> obtained from <code>IDbCommand</code>.

20.6.3. Execute Callback in .NET 1.1

>

AdoTemplate differs from its .NET 2.0 generic counterpart in that it exposes the interface <code>IDbCommand</code> in its 'Execute' callback methods and delegate as compared to the abstract base class <code>DbProvider</code>. Also, since anonymous delegates are not available in .NET 1.1, the typical usage pattern requires you to

create a explicitly delegate and/or class that implements the **ICommandCallback** interface. Example code to query In .NET 1.1 the 'Northwind' database is done to determine the number of customers who have a particular postal code is shown below.

```
public virtual int FindCountWithPostalCode(string postalCode
{
  return (int) AdoTemplate.Execute(new PostalCodeCommandCall
}
```

and the callback implementation is

```
private class PostalCodeCommandCallback : ICommandCallback
{
    private string cmdText = "select count(*) from Customer
    private string postalCode;
    public PostalCodeCommandCallback(string postalCode)
    {
        this.postalCode = postalCode;
    }
    public object DoInCommand(IDbCommand command)
        command.CommandText = cmdText;
        IDbDataParameter p = command.CreateParameter();
        p.ParameterName = "@PostalCode";
        p.Value = postalCode;
        command.Parameters.Add(p);
        return command.ExecuteScalar();
    }
}
```

Note that in this example, one could more easily use AdoTemplate's ExecuteScalar method.

The Execute method has interface and delegate overloads. The signatures for the

delegate and interface are shown below

```
public delegate object CommandDelegate(IDbCommand command);
public interface ICommandCallback
{
   object DoInCommand(IDbCommand command);
}
```

The corresponding method signatures on Spring.Data.AdoTemplate are shown below

```
public class AdoTemplate : AdoAccessor, IAdoOperations
{
    ...
    object Execute(CommandDelegate del);
    object Execute(ICommandCallback action);
    ...
}
```

Note that you have to cast to the appropriate object type returned from the execute method.

20.6.4. Quick Guide to AdoTemplate Methods

There are many methods in AdoTemplate so it is easy to feel a bit overwhelmed when taking a look at the SDK documentation. However, after a while you will hopefully find the class 'easy to navigate' with intellisense. Here is a quick categorization of the method names and their associated data access operation. Each method is overloaded to handle common cases of passing in parameter values.

The generic 'catch-all' method

• Execute - Allows you to perform any data access operation on a standard DbCommand object. The connection and transaction properties of the DbCommand are already set based on the transactional calling context.

There is also an overloaded method that operates on a standard IDbCommand object. This is for those providers that do not inherit from the base class DbCommand.

The following methods mirror those on the DbCommand object.

- ExecuteNonQuery Executes the 'NonQuery' method on a DbCommand, applying provided parameters and returning the number of rows affected.
- ExecuteScalar Executes the 'Scalar' method on a DbCommand, applying provided parameters, and returning the first column of the first row in the result set.

Mapping result sets to objects

- QueryWithResultSetExtractor Execute a query mapping a result set to an object with an implementation of the IResultSetExtractor interface.
- QueryWithResultSetExtractorDelegate Same as QueryWithResultSetExtractor but using a ResultSetExtractorDelegate to perform result set mapping.
- QueryWithRowCallback Execute a query calling an implementation of IRowCallback for each row in the result set.
- QueryWithRowCallbackDelegate Same as QueryWithRowCallback but calling a RowCallbackDelegate for each row.
- QueryWithRowMapper Execute a query mapping a result set on a row by row basis with an implementation of the IRowMapper interface.
- QueryWithRowMapperDelegate Same as QueryWithRowMapper but using a RowMapperDelegate to perform result set row to object mapping.

Mapping result set to a single object

• QueryForObject - Execute a query mapping the result set to an object using a IRowMapper. Exception is thrown if the query does not return exactly one object.

Query with a callback to create the DbCommand object. These are generally used by the framework itself to support other functionality, such as in the Spring.Data.Objects namespace.

• QueryWithCommandCreator - Execute a query with a callback to IDbCommandCreator to create a IDbCommand object and using either a IRowMapper or IResultSetExtractor to map the result set to an object. One variation lets multiple result set 'processors' be specified to act on multiple result sets and return output parameters.

DataTable and DataSet operations

- DataTableCreate Create and Fill DataTables
- DataTableCreateWithParameters Create and Fill DataTables using a parameter collection.
- DataTableFill Fill a pre-existing DataTable.
- DataTableFillWithParameters Fill a pre-existing DataTable using parameter collection.
- DataTableUpdate Update the database using the provided DataTable, insert, update, delete SQL.
- DataTableUpdateWithCommandBuilder Update the database using the provided DataTable, select SQL, and parameters.
- DataSetCreate Create and Fill DataSets
- DataSetCreateWithParameters Create and Fill DataTables using a parameter collection.
- DataSetFill Fill a pre-existing DataSet
- DataSetFillWithParameters Fill a pre-existing DataTable using parameter collection.
- DataSetUpdate Update the database using the provided DataSet, insert, update, delete SQL.
- DataSetUpdateWithCommandBuilder Update the database using the provided DataSet, select SQL, and parameters..



These methods are not currently in the generic version of AdoTemplate but accessible through the property ClassicAdoTemplate.

Parameter Creation utility methods

• DeriveParameters - Derive the parameter collection for stored procedures.

In turn each method typically has four overloads, one with no parameters and three for providing parameters. Aside from the DataTable/DataSet operations, the three parameter overloads are of the form shown below

• MethodName(CommandType cmdType, string cmdText, CallbackInterfaceOrDelegate, parameter setting arguments)

The CallbackInterfaceOrDelegate is one of the three types listed previously. The parameters setting arguments are of the form

- MethodName(... string parameterName, Enum dbType, int size, object parameterValue)
- MethodName(... IDbParameters parameters)
- MethodName(... ICommandSetter commandSetter)

The first overload is a convenience method when you only have one parameter to set. The database enumeration is the base class 'Enum' allowing you to pass in any of the provider specific enumerations as well as the common DbType enumeration. This is a trade off of type-safety with provider portability. (Note generic version could be improved to provide type safety...).

The second overload contains a collection of parameters. The data type is Spring's IDbParameters collection class discussed in the following section.

The third overload is a callback interface allowing you to set the parameters (or other properties) of the IDbCommand passed to you by the framework directly.

If you are using .NET 2.0 the delegate versions of the methods are very useful since very compact definitions of database operations can be created that reference variables local to the DAO method. This removes some of the tedium in passing parameters around with interface based versions of the callback functions since they need to be passed into the constructor of the implementing class. The general guideline is to use the delegate when available for

functionality that does not need to be shared across multiple DAO classes or methods and use interface based version to reuse the implementation in multiple places. The .NET 2.0 versions make use of generics where appropriate and therefore enhance type-safety.

20.6.5. Quick Guide to AdoTemplate Properties

AdoTemplate has the following properties that you can configure

- LazyInit Indicates if the IAdoExceptionTranslator should be created on first encounter of an exception from the data provider or when AdoTemplate is created. Default is true, i.e. to lazily instantiate.
- ExceptionTranslator Gets or sets the implementation of IAdoExceptionTranslator to use. If no custom translator is provided, a default ErrorCodeExceptionTranslator is used.
- DbProvider Gets or sets the IDbProvider instance to use.
- DataReaderWrapperType Gets or set the System. Type to use to create an instance of IDataReaderWrapper for the purpose of providing mapping functionality. extended Spring implementation to use as the basis for a mapping strategy that will map **DBNull** values default values based the standard to on **IDataReader** interface. See the section custom IDataReader implementations for more information.
- **CommandTimeout** Gets or sets the command timeout for IDbCommands that this **AdoTemplate** executes. Default is 0, indicating to use the database provider's default.

20.7. Transaction Management

The AdoTemplate is used in conjunction with an implementation of a IPlatformTransactionManager, which is Spring's portable transaction management API. This section gives a brief overview of the transaction managers you can use with AdoTemplate and the details of how you can retrieve the connection/transaction ADO.NET objects that are bound to the thread when a transaction starts. Please refer to the section key abstractions in the chapter on transactions for more comprehensive introduction to transaction management.

To use local transactions, those with only one transactional resource (i.e. the database) you will typically use AdoPlatformTransactionManager. If you need to mix Hibernate and ADO.NET data access operations within the same local transaction you should use HibernatePlatformTransaction manager which is described more in the section on ORM transaction management.

While it is most common to use Spring's <u>transaction management features</u> to avoid the low level management of ADO.NET connection and transaction objects, you can retrieve the connection/transaction pair that was created at the start of a transaction and bound to the current thread. This may be useful for some integration with other data access APIs. The can be done using the utility class ConnectionUtils as shown below.

```
IDbProvider dbProvider = DbProviderFactory.GetDbProvider("Sy
ConnectionTxPair connectionTxPairToUse = ConnectionUtils.Get

IDbCommand command = DbProvider.CreateCommand();
command.Connection = connectionTxPairToUse.Connection;
command.Transaction = connectionTxPairToUse.Transaction;
```

It is possible to provide a wrapper around the standard .NET provider interfaces such that you can use the plain ADO.NET API in conjunction with Spring's transaction management features.

```
If you are using ServiceDomainPlatformTransactionManager or
```

TxScopePlatformTransactionManager then you can retrieve the currently executing transaction object via the standard .NET APIs.

20.8. Exception Translation

AdoTemplate's methods throw exceptions within a Data Access Object (DAO) exception hierarchy described in Chapter 18, DAO support. In addition, the command text and error code of the exception are extracted and logged. This leads to easier to write provider independent exception handling layer since the exceptions thrown are not tied to a specific persistence technology. Additionally, for ADO.NET code the error messages logged provide information on the SQL and error code to better help diagnose the issue.

20.9. Parameter Management

A fair amount of the code in ADO.NET applications is related to the creation and population of parameters. The BCL parameter interfaces are very minimal and do not have many convenience methods found in provider implementations such as SqlClient. Even still, with SqlClient, there is a fair amount of verbosity to creating and populating a parameter collection. Spring provides two ways to make this mundane task easier and more portable across providers.

20.9.1. IDbParametersBuilder

Instead of creating a parameter on one line of code, then setting its type on another and size on another, a builder and parameter interface, IDbParametersBuilder and IDbParameter respectfully, are provided so that this declaration process can be condensed. The IDbParameter support chaining calls to its methods, in effect a simple language-constrained domain specific language, to be fancy about it. Here is an example of it in use.

```
IDbParametersBuilder builder = CreateDbParametersBuilder();
builder.Create().Name("Country").Type(DbType.String).Size(15)
builder.Create().Name("City").Type(DbType.String).Size(15).V

// now get the IDbParameters collection for use in passing to IDbParameters parameters = builder.GetParameters();
```

Please note that IDbParameters and IDbParameter are not part of the BCL, but part of the Spring.Data.Common namespace. The IDbParameters collection is a frequent argument to the overloaded methods of AdoTemplate.

The parameter prefix, i.e. '@' in Sql Server, is not required to be added to the parameter name. The DbProvider is aware of this metadata and AdoTemplate will add it automatically if required before execution.

An additional feature of the IDbParametersBuilder is to create a Spring FactoryObject that creates IDbParameters for use in the XML configuration file of the IoC container. By leveraging Spring's expression evaluation language, the above lines of code can be taken as text from the XML configuration file and

executed. As a result you can externalize your parameter definitions from your code. In combination with abstract object definitions and importing of configuration files your increase the chances of having one code base support multiple database providers just by a change in configuration files.

20.9.2. IDbParameters

This class is similar to the parameter collection class you find in provider specific implementations of IDataParameterCollection. It contains a variety of convenience methods to build up a collection of parameters.

Here is an abbreviated listing of the common convenience methods.

- int Add(object parameterValue)
- void AddRange(Array values)
- IDbDataParameter AddWithValue(string name, object parameterValue)
- IDbDataParameter Add(string name, Enum parameterType)
- IDbDataParameter AddOut(string name, Enum parameterType)
- IDbDataParameter AddReturn(string name, Enum parameterType)
- void DeriveParameters(string storedProcedureName)

Here a simple usage example

```
// inside method has has local variable country and city...
IDbParameters parameters = CreateDbParameters();
parameters.AddWithValue("Country", country).DbType = DbType.
parameters.Add("City", DbType.String).Value = city;
// now pass on to AdoTemplate methods.
```

The parameter prefix, i.e. '@' in Sql Server, is not required to be added to the parameter name. The DbProvider is aware of this metadata and AdoTemplate will add it automatically if required before execution.

20.10. Custom IDataReader implementations

The passed in implementation of <code>IDataReader</code> can be customized. This lets you add a strategy for handling null values to the standard methods in the <code>IDataReader</code> interface or to provide sub-interface of <code>IDataReader</code> that contains extended functionality, for example support for default values. In callback code, i.e. <code>IRowMapper</code> and associated delegate, you would downcast to the sub-interface to perform processing.

Spring provides a class to map DBNull values to default values. When reading from a IDataReader there is often the need to map DBNull values to some default values, i.e. null or say a magic number such as -1. This is usually done via a ternary operator which decreases readability and also increases the likelihood of mistakes. Spring provides an IDataReaderWrapper interface (which inherits from the standard IDataReader) so that you can provide your own implementation of a IDataReader that will perform DBNull mapping for you in a consistent and non invasive manner to your result set reading code. A default implementation, NullMappingDataReader is provided which you can subclass to customize or simply implement the IDataReaderWrapper interface directly. This interface is shown below

```
public interface IDataReaderWrapper : IDataReader
{
    IDataReader WrappedReader
        {
            get;
            set;
        }
}
```

All callback interfaces/delegates AdoTemplates have an **IDataReader** an argument are wrapped with a IDataReaderWrapper if the AdoTemplate has been configured with one via its DataReaderWrapperType property. Your implementation should support a zero-arg constructor.

Frequently you will use a common mapper for DBNull across your application

so only one instance of AdoTemplate and IDataReaderWrapper in required. If you need to use multiple null mapping strategies you will need to create multiple instances of AdoTemplate and configure them appropriately in the DAO objects.

20.11. Basic data access operations

The 'ExecuteNonQuery' and 'ExecuteScalar' methods of AdoTemplate have the same functionality as the same named methods on the DbCommand object

20.11.1. ExecuteNonQuery

ExecuteNonQuery is used to perform create, update, and delete operations. It has four overloads listed below reflecting different ways to set the parameters.

An example of using this method is shown below

20.11.2. ExecuteScalar

An example of using this method is shown below

```
int iCount = (int)adoTemplate.ExecuteScalar(CommandType.Text
```

20.12. Queries and Lightweight Object Mapping

A common ADO.NET development task is reading in a result set and converting it to a collection of domain objects. The family of QueryWith methods on AdoTemplate help in this task. The responsibility of performing the mapping is given to one of three callback interfaces/delegates that you are responsible for developing. These callback interfaces/delegates are:

- IResultSetExtractor / ResultSetExtractorDelegate hands you a IDataReader object for you to iterate over and return a result object.
- IRowCallback / RowCallbackDelegate hands you a IDataReader to process the current row. Returns void and as such is usually stateful in the case of IRowCallback implementations or uses a variable to collect a result that is available to an anonymous delegate.
- IRowMapper / RowMapperDelegate hands you a IDataReader to process the current row and return an object corresponding to that row.

There are generic versions of the IResultSetExtractor and IRowMapper interfaces/delegates providing you with additional type-safety as compared to the object based method signatures used in the .NET 1.1 implementation.

As usual with callback APIs in Spring.Data, your implementations of these interfaces/delegates are only concerned with the core task at hand - mapping data - while the framework handles iteration of readers and resource management.

Each 'QueryWith' method has 4 overloads to handle common ways to bind parameters to the command text.

The following sections describe in more detail how to use Spring's lightweight object mapping framework.

20.12.1. ResultSetExtractor

The ResultSetExtractor gives you control to iterate over the IDataReader returned from the query. You are responsible for iterating through all the result sets and returning a corresponding result object. Implementations of IResultSetExtractor are typically stateless and therefore reusable as long as the implementation doesn't access stateful resources. The framework will close the IDataReader for you.

The interface and delegate signature for ResutSetExtractors is shown below for

the generic version in the Spring.Data.Generic namespace

```
public interface IResultSetExtractor<T>
{
   T ExtractData(IDataReader reader);
}

public delegate T ResultSetExtractorDelegate<T>(IDataReader
```

The definition for the non-generic version is shown below

```
public interface IResultSetExtractor
{
    object ExtractData(IDataReader reader);
}
public delegate object ResultSetExtractorDelegate(IDataReader)
```

Here is an example taken from the Spring.DataQuickStart. It is a method in a DAO class that inherits from AdoDaoSupport, which has a convenience method 'CreateDbParametersBuilder()'.

```
public virtual IList<string> GetCustomerNameByCountr
{

    IDbParametersBuilder builder = CreateDbParameter
    builder.Create().Name("Country").Type(DbType.Str
    builder.Create().Name("City").Type(DbType.String
    return AdoTemplate.QueryWithResultSetExtractor(Country)
}
```

The implementation of the ResultSetExtractor is shown below.

```
internal class CustomerNameResultSetExtractor<T> : IResu
{
    public T ExtractData(IDataReader reader)
```

```
{
    T customerList = new T();
    while (reader.Read())
    {
        string contactName = reader.GetString(0);
        customerList.Add(contactName);
    }
    return customerList;
}
```

of Internally the implementation the QueryWithRowCallback and QueryWithRowMapper specializations methods are of the ResultSetExtractor. For example, the QueryWithRowMapper implementation iterates through the result set, calling the callback method 'MapRow' for each row and collecting the results in an IList. If you have a specific case that is not covered by the QueryWithXXX methods you can subclass AdoTemplate and follow the same implementation pattern to create a new QueryWithXXX method to suit your needs.

20.12.2. RowCallback

The RowCallback is usually a stateful object itself or populates another stateful object that is accessible to the calling code. Here is a sample take from the Data QuickStart

The PostalCodeRowCallback builds up state which is then retrieved via the property PostalCodeMultimap. The Callback implementation is shown below

```
internal class PostalCodeRowCallback : IRowCallback
{
    private IDictionary<string, IList<string>> postalCod
        new Dictionary<string, IList<string>>();
    public IDictionary<string, IList<string>> PostalCode
        get { return postalCodeMultimap; }
    public void ProcessRow(IDataReader reader)
        string contactName = reader.GetString(0);
        string postalCode = reader.GetString(1);
        IList<string> contactNameList;
        if (postalCodeMultimap.ContainsKey(postalCode))
            contactNameList = postalCodeMultimap[postal(
        else
            postalCodeMultimap.Add(postalCode, contactNa
        contactNameList.Add(contactName);
    }
}
```

20.12.3. RowMapper

The RowMapper lets you focus on just the logic to map a row of your result set to an object. The creation of a IList to store the results and iterating through the IDataReader is handled by the framework. Here is a simple example taken from the Data QuickStart application

where the implementation of the RowMapper is

```
public class CustomerRowMapper<T> : IRowMapper<T> where
{
    public T MapRow(IDataReader dataReader, int rowNum)
        T customer = new T();
        customer.Address = dataReader.GetString(0);
        customer.City = dataReader.GetString(1);
        customer.CompanyName = dataReader.GetString(2);
        customer.ContactName = dataReader.GetString(3);
        customer.ContactTitle = dataReader.GetString(4);
        customer.Country = dataReader.GetString(5);
        customer.Fax = dataReader.GetString(6);
        customer.Id = dataReader.GetString(7);
        customer.Phone = dataReader.GetString(8);
        customer.PostalCode = dataReader.GetString(9);
        customer.Region = dataReader.GetString(10);
        return customer;
    }
}
```

You may also pass in a delegate, which is particularly convenient if the mapping logic is short and you need to access local variables within the mapping logic.

```
customer.Address = dataReade
customer.City = dataReader.G
customer.CompanyName = dataF
customer.ContactName = dataF
customer.ContactTitle = data
customer.Country = dataReade
customer.Fax = dataReader.Ge
customer.Id = dataReader.Get
customer.Phone = dataReader.
customer.PostalCode = dataRe
customer.Region = dataReader
return customer;
});
```

20.12.4. Query for a single object

The QueryForObject method is used when you expect there to be exactly one object returned from the mapping, otherwise a Spring.Dao.IncorrectResultSizeDataAccessException will be thrown. Here is some sample usage taken from the Data QuickStart.

```
public class QueryForObjectDao : AdoDaoSupport
{
    private string cmdText = "select Address, City, Compount "ContactTitle, Country, Fax, CustomerID" "Region from Customers where ContactNam

    public Customer GetCustomer(string contactName)
    {
        return AdoTemplate.QueryForObject(CommandType.Temple new CustomerRomer "ContactName",
        }
}
```

20.12.5. Query using a CommandCreator

There is a family of overloaded methods that allows you to encapsulate and reuse a particular configuration of a **IDbCommand** object. These methods

also allow for access to returned out parameters as well as a method that allows processing of multiple result sets. These methods are used internally to support the classes in the Spring.Data.Objects namespace and you may find the API used in that namespace to be more convenient. The family of methods is listed below.

- object QueryWithCommandCreator(IDbCommandCreator cc, IResultSetExtractor rse)
- void
 QueryWithCommandCreator(IDbCommandCreator
 cc, IRowCallback rowCallback)
- IList QueryWithCommandCreator(IDbCommandCreator cc, IRowMapper rowMapper)

There is also the same methods with an additional collecting parameter to obtain any output parameters. These are

- object QueryWithCommandCreator(IDbCommandCreator cc, IResultSetExtractor rse, IDictionary returnedParameters)
- void QueryWithCommandCreator(IDbCommandCreator cc, IRowCallback rowCallback, IDictionary returnedParameters)
- IList
 QueryWithCommandCreator(IDbCommandCreator
 cc, IRowMapper rowMapper, IDictionary
 returnedParameters)

The IDbCommandCreator callback interface is shown below

```
public interface IDbCommandCreator
{
     IDbCommand CreateDbCommand();
}
```

The created IDbCommand object is used when performing the QueryWithCommandCreator method.

To process multiple result sets specify a list of named result set processors, (i.e. IResultSetExtractor, IRowCallback, or IRowMapper). This method is shown below

 IDictionary QueryWithCommandCreator(IDbCommandCreator cc, IList namedResultSetProcessors)

The list must contain objects of the type Spring.Data.Support.NamedResultSetProcessor. This is the class responsible for associating a name with a result set processor. The constructors are listed below.

```
public class NamedResultSetProcessor {
  public NamedResultSetProcessor(string name, IRowMapper row
  public NamedResultSetProcessor(string name, IRowCallback r
  public NamedResultSetProcessor(string name, IResultSetExtr
    . . .
}
```

The results of the RowMapper or ResultSetExtractor are retrieved by name from the dictionary that is returned. RowCallbacks, being stateless, only have the placeholder text, "ResultSet returned was processed by an IRowCallback" as a value for the name of the RowCallback used as a key. Output and InputOutput parameters can be retrieved by name. If this parameter name is null, then the index of the parameter prefixed with the letter 'P' is a key name, i.e P2, P3, etc.

The namespace Spring.Data.Objects.Generic contains generic versions of these methods. These are listed below

T QueryWithCommandCreator<T>
 (IDbCommandCreator cc,

IResultSetExtractor<T> rse)

IList<T> QueryWithCommandCreator<T>
 (IDbCommandCreator cc, IRowMapper<T>
 rowMapper)

and overloads that have an additional collecting parameter to obtain any output parameters.

- T QueryWithCommandCreator<T>
 (IDbCommandCreator cc,
 IResultSetExtractor<T> rse, IDictionary
 returnedParameters)
- IList<T> QueryWithCommandCreator<T>
 (IDbCommandCreator cc, IRowMapper<T>
 rowMapper, IDictionary returnedParameters)

When processing multiple result sets you can specify up to two type safe result set processors.

- IDictionary QueryWithCommandCreator<T>
 (IDbCommandCreator cc, IList namedResultSetProcessors)
- IDictionary QueryWithCommandCreator<T,U>
 (IDbCommandCreator cc, IList namedResultSetProcessors)

The list of result set processors contains either objects of the type Spring.Data.Generic.NamedResultSetProcessor<T> Spring.Data.NamedResultSetProcessor. The generic result set processors, NamedResultSetProcessor<T>, is used to process the first result set in the case of using QueryWithCommandCreator<T> and to process the first and second result set in the case of using QueryWithCommandCreator<T,U>. Additional Spring.Data.NamedResultSetProcessors that are listed can be used to process additional RowCallback result sets. If you specify a NamedResultSetProcessor<T>, you still need to specify a type parameter (say string) because the RowCallback processor does not return any object. It is up to subclasses of RowCallback to collect state due to processing the result set which is later queried.

20.13. DataTable and DataSet

AdoTemplate contains several 'families' of methods to help remove boilerplate code and reduce common programming errors when using DataTables and DataSets. There are many methods in AdoTemplate so it is easy to feel a bit overwhelmed when taking a look at the SDK documentation. However, after a while you will hopefully find the class 'easy to navigate' with intellisense. Here is a quick categorization of the method names and their associated data access operation. Each method is overloaded to handle common cases of passing in parameter values.

The 'catch-all' Execute methods upon which other functionality is built up upon are shown below.

In Spring.Data.Core.AdoTemplate

• object Execute(IDataAdapterCallback dataAdapterCallback) - Execute ADO.NET operations on a IDbDataAdapter object using an interface based callback.

Where IDataAdapterCallback is defined as

```
public interface IDataAdapterCallback
{
    object DoInDataAdapter(IDbDataAdapter dataAdapter);
}
```

The passed in **IDbDataAdapter** will have its **SelectCommand** property created and set with its **Connection** and **Transaction** values based on the calling transaction context. The return value is the result of processing or null.

There are type-safe versions of this method in Spring.Data.Generic.AdoTemplate

- T Execute<T>(IDataAdapterCallback<T> dataAdapterCallback) Execute ADO.NET operations on a IDbDataAdapter object using an interface based callback.
- T Execute<T>(DataAdapterDelegate<T> del) Execute ADO.NET operations on a IDbDataAdapter object using an delegate based callback.

```
public interface IDataAdapterCallback<T>
{
    T DoInDataAdapter(IDbDataAdapter dataAdapter);
}
public delegate T DataAdapterDelegate<T>(IDbDataAdapter data
```

20.13.1. DataTables

DataTable operations are available on the class Spring.Data.Core.AdoTemplate. If you are using the generic version, Spring.Data.Generic.AdoTemplate, you can access these methods through the property ClassicAdoTemplate, which returns the non-generic version of AdoTemplate. DataTable operations available fall into the general family of methods with 3-5 overloads per method.

- DataTableCreate Create and Fill DataTables
- DataTableCreateWithParameters Create and Fill DataTables using a parameter collection.
- DataTableFill Fill a pre-existing DataTable.
- DataTableFillWithParameters Fill a pre-existing DataTable using a parameter collection.
- DataTableUpdate Update the database using the provided DataTable, insert, update, delete SQL.
- DataTableUpdateWithCommandBuilder Update the database using the provided DataTable, select SQL, and parameters.

20.13.2. DataSets

DataSet operations are available on the class Spring.Data.Core.AdoTemplate. If you are using the generic version, Spring.Data.Generic.AdoTemplate, you can access these methods through the property ClassicAdoTemplate, which returns the non-generic version of AdoTemplate. DataSet operations available fall into the following family of methods with 3-5 overloads per method.

- DataSetCreate Create and Fill DataSets
- DataSetCreateWithParameters Create and Fill DataTables using a parameter collection.
- DataSetFill Fill a pre-existing DataSet
- DataSetFillWithParameters Fill a pre-existing DataTable using parameter collection.
- DataSetUpdate Update the database using the provided DataSet, insert, update, delete SQL.
- DataSetUpdateWithCommandBuilder Update the database using the provided DataSet, select SQL, and parameters.

The following code snippets demonstrate the basic functionality of these methods using the Northwind database. See the SDK documentation for more details on other overloaded methods.

```
"ContactTitle, Country, Fax, Custon
"Region from Customers where Contac

DbParameters dbParameters = CreateDbParameters();
dbParameters.Add("ContactName", DbType.String).Value = 'DataSet customerLikeMDataSet = AdoTemplate.DataSetCreate

// customerLikeMDataSet has a table named 'Table' with 12
}

public void DemoDataSetFill()
{
   DataSet dataSet = new DataSet();
   dataSet.Locale = CultureInfo.InvariantCulture;
   AdoTemplate.DataSetFill(dataSet, CommandType.Text, select)
```

Updating a DataSet can be done using a CommandBuilder, automatically created from the specified select command and select parameters, or by explicitly specifying the insert, update, delete commands and parameters. Below is an example, refer to the SDK documentation for additional overloads

```
dataSet.Locale = CultureInfo.InvariantCulture;
    AdoTemplate.DataSetFill(dataSet, CommandType.Text, selection)
    AddAndEditRow(dataSet);.
    string insertSql = @"INSERT Customers (CustomerID, Compa
    IDbParameters insertParams = CreateDbParameters();
    insertParams.Add("CustomerId", DbType.String, 0, "Custom
    insertParams.Add("CompanyName", DbType.String, 0, "Compa
    string updateSql = @"update Customers SET Phone=@Phone w
    IDbParameters updateParams = CreateDbParameters();
    updateParams.Add("Phone", DbType.String, 0, "Phone");//.
    updateParams.Add("CustomerId", DbType.String, 0, "Custom
    AdoTemplate.DataSetUpdate(dataSet, "Customers",
                                         CommandType.Text, in
                                         CommandType.Text, up
                                         CommandType.Text, nu
 }
  private static void AddAndEditRow(DataSet dataSet)
    DataRow dataRow = dataSet.Tables["Customers"].NewRow();
    dataRow["CustomerId"] = "NewID";
    dataRow["CompanyName"] = "New Company Name";
    dataRow["ContactName"] = "New Name";
    dataRow["ContactTitle"] = "New Contact Title";
    dataRow["Address"] = "New Address";
    dataRow["City"] = "New City";
    dataRow["Region"] = "NR";
    dataRow["PostalCode"] = "New Code";
    dataRow["Country"] = "New Country";
    dataRow["Phone"] = "New Phone";
    dataRow["Fax"] = "New Fax";
    dataSet.Tables["Customers"].Rows.Add(dataRow);
    DataRow alfkiDataRow = dataSet.Tables["Customers"].Rows[
    alfkiDataRow["Phone"] = "030-0074322"; // simple change,
 }
}
```

In the case of needing to set parameter SourceColumn or SourceVersion properties it may be more convenient to use IDbParameterBuilder.

20.14. TableAdapters and participation in transactional context

Typed DataSets need to have commands in their internal DataAdapters and command collections explicitly set with a connection/transaction in order for them to correctly participate with a surrounding transactional context. The reason for this is by default the code generated is explicitly managing the connections and transactions. This issue is very well described in the article System.Transactions and ADO.NET 2.0 by ADO.NET guru Sahil Malik. Spring offers a convenience method that will use reflection to internally set the transaction on the table adapter's internal command collection to the ambient transaction. This method on the class Spring.Data.Support.TypedDataSetUtils and is named ApplyConnectionAndTx. Here is sample usage of a DAO method that

uses a VS.NET 2005 generated typed dataset for a PrintGroupMapping table.

This DAO method may be combined with other DAO operations inside a transactional context and they will all share the same connection/transaction objects.

There are two overloads of the method ApplyConnectionAndTx which differ in

the second method argument, one takes an IDbCommand and the other IDbProvider. These are listed below

```
public static void ApplyConnectionAndTx(object typedDataSetA
public static void ApplyConnectionAndTx(object typedDataSetA
```

The method that takes IDbCommand is a convenience if you will be using AdoTemplate callback's as the passed in command object will already have its connection and transaction properties set based on the current transactional context. The method that takes an IDbProvider is convenient to use when you have data access logic that is not contained within a single callback method but is instead spead among multiple classes. In this case passing the transactionally aware IDbCommand object can be intrusive on the method signatures. Instead you can pass in an instance of IDbProvider that can be obtained via standard dependency injection techniques or via a service locator style lookup.

20.15. Database operations as Objects

The Spring.Data.Objects

and

Spring.Data.Objects.Generic namespaces contains classes that allow one to access the database in a more object-oriented manner. By way of an example, one can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object. One can also execute stored procedures and run update, delete and insert statements.



Note

There is a view borne from experience acquired in the field amongst some of the Spring developers that the various RDBMS operation classes described below (with the exception of the StoredProcedure class) can often be replaced with straight AdoTemplate calls... often it is simpler to use and plain easier to read a DAO method that simply calls a method on a AdoTemplate direct (as opposed to encapsulating a query as a full-blown class).

It must be stressed however that this is just a *view*... if you feel that you are getting measurable value from using the RDBMS operation classes, feel free to continue using these classes.

20.15.1. AdoQuery

AdoQuery is a reusable, threadsafe class that encapsulates an SQL query. Subclasses must implement the NewRowMapper (. .) method to provide a **IROWMapper** instance that can create one object per row obtained from iterating over the IDataReader that is created during the execution of the The AdoQuery class rarely is used directly since MappingAdoQuery subclass provides a much more convenient implementation for mapping rows to .NET classes. Another implementations that extends AdoQuery is MappingadoQueryWithParameters (See SDK docs for details).

The AdoNonQuery class encapsulates an IDbCommand 's ExecuteNonQuery method functionality. Like the AdoQuery object, an AdoNonQuery object is reusable, and like all AdoOperation classes,

an AdoNonQuery can have parameters and is defined in SQL. This class provides two execute methods

- IDictionary ExecuteNonQuery(params object[] inParameterValues)
- IDictionary ExecuteNonQueryByNamedParam(IDictionary inParams)

This class is concrete. Although it can be subclassed (for example to add a custom update method) it can easily be parameterized by setting SQL and declaring parameters.

An example of an AdoQuery subclass to encapsulate an insert statement for a 'TestObject' (consisting only name and age columns) is shown below

```
public class CreateTestObjectNonQuery : AdoNonQuery
{
  private static string sql = "insert into TestObjects(Age,N)
  public CreateTestObjectNonQuery(IDbProvider dbProvider) :
    {
     DeclaredParameters.Add("Age", DbType.Int32);
     DeclaredParameters.Add("Name", SqlDbType.NVarChar, 16);
     Compile();
  }
  public void Create(string name, int age)
  {
     ExecuteNonQuery(name, age);
  }
}
```

20.15.2. MappingAdoQuery

MappingAdoQuery is a reusable query in which concrete subclasses must implement the abstract MapRow(..) method to convert each row of the supplied IDataReader into an object. Find below a brief example of a custom query that maps the data from a relation to an instance of the

Customer class.

```
public class TestObjectQuery : MappingAdoQuery
{
    private static string sql = "select TestObjectNo, Age, N
    public TestObjectQuery(IDbProvider dbProvider)
        : base(dbProvider, sql)
    {
        CommandType = CommandType.Text;
    }

    protected override object MapRow(IDataReader reader, int
    {
        TestObject to = new TestObject();
        to.ObjectNumber = reader.GetInt32(0);
        to.Age = reader.GetInt32(1);
        to.Name = reader.GetString(2);
        return to;
    }
}
```

20.15.3. AdoNonQuery

The AdoNonQuery class encapsulates an IDbCommand 's ExecuteNonQuery method functionality. Like the AdoQuery object, an AdoNonQuery object is reusable, and like all AdoOperation classes, an AdoNonQuery can have parameters and is defined in SQL. This class provides two execute methods

- IDictionary ExecuteNonQuery(params object[] inParameterValues)
- IDictionary ExecuteNonQueryByNamedParam(IDictionary inParams)

This class is concrete. Although it can be subclassed (for example to add a custom update method) it can easily be parameterized by setting SQL and declaring parameters.

```
public class CreateTestObjectNonQuery : AdoNonQuery
{
  private static string sql = "insert into TestObjects(Age,N)
  public CreateTestObjectNonQuery(IDbProvider dbProvider) :
    {
      DeclaredParameters.Add("Age", DbType.Int32);
      DeclaredParameters.Add("Name", SqlDbType.NVarChar, 16);
      Compile();
    }
  public void Create(string name, int age)
    {
      ExecuteNonQuery(name, age);
    }
}
```

20.15.4. Stored Procedure

The StoredProcedure class is designed to make it as simple as possible to call a stored procedure. It takes advantage of metadata present in the database to look up names of in and out parameters.. This means that you don't have to explicitly declare parameters. You can of course still declare them if you prefer. There are two versions of the StoredProcedure class, one that uses generics and one that doesn't. Using the StoredProcedure class consists of two steps, first defining the in/out parameter and any object mappers and second executing the stored procedure.

The non-generic version of StoredProcedure is in the namespace Spring.Data.Objects. It contains the following methods to execute a stored procedure

- IDictionary ExecuteScalar(params object[] inParameterValues)
- IDictionary ExecuteScalarByNamedParam(IDictionary inParams)
- IDictionary ExecuteNonQuery(params object[]

inParameterValues)

- IDictionary ExecuteNonQueryByNamedParam(IDictionary inParams)
- IDictionary Query(params object[] inParameterValues)
- IDictionary QueryByNamedParam(IDictionary inParams)

Each of these methods returns an **IDictionary** that contains the output parameters and/or any results from Spring's object mapping framework. The arguments to these methods can be a variable length argument list, in which case the order must match the parameter order of the stored procedure. If the argument is an IDictionary it contains parameter key/value pairs. Return values from stored procedures are contained under the key "RETURN_VALUE".

The standard in/out parameters for the stored procedure can be set programmatically by adding to the parameter collection exposed by the property DeclaredParameters. For each result sets that is returned by the stored procedures you can registering either an IResultSetExtractor, IRowCallback, or IRowMapper by name, which is used later to extract the mapped results from the returned IDictionary.

Lets take a look at an example. The following stored procedure class will call the CustOrdersDetail stored procedure in the Northwind database, passing in the OrderID as a stored procedure argument and returning a collection of OrderDetails business objects.

```
public class CustOrdersDetailStoredProc : StoredProcedur
{
    private static string procedureName = "CustOrdersDet

    public CustOrdersDetailStoredProc(IDbProvider dbProv
    {
        DeriveParameters();
        AddRowMapper("orderDetailRowMapper", new OrderDetaile();
        }
}
```

```
public virtual IList GetOrderDetails(int orderid)
{
    IDictionary outParams = Query(orderid);
    return outParams["orderDetailRowMapper"] as ILis
}
```

The 'DeriveParameters' method saves you the trouble of having to declare each parameter explicitly. When using DeriveParameters is it often common to use the Query method that takes a variable length list of arguments. This assumes additional knowledge on the order of the stored procedure arguments. If you do not want to follow this loose shorthand convention, you can call the method QueryByNamesParameters instead passing in a IDictionary of parameter key/value pairs.



Note

If you would like to have the return value of the stored procedure included in the returned dictionary, pass in true as a method parameter to DeriveParameters().

The StoredProcedure class is threadsafe once 'compiled', an act which is usually done in the constructor. This sets up the cache of database parameters that can be used on each call to Query or QueryByNamedParam. The implementation of IROwMapper that is used to extract the business objects is 'registered' with the class and then later retrieved by name as a fictional output IRowCallback and parameter. You may also register **IResultSetExtractor** callback interfaces via the AddRowCallback and AddResultSetExtractor methods.

The generic version of StoredProcedure is in the namespace Spring.Data.Objects.Generic. It allows you to define up to two generic type parameters that will be used to process result sets returned from the stored procedure. An example is shown below

```
public class CustOrdersDetailStoredProc : StoredProcedur
{
```

```
private static string procedureName = "CustOrdersDet

public CustOrdersDetailStoredProc(IDbProvider dbProv
{
    DeriveParameters();
    AddRowMapper("orderDetailRowMapper", new OrderDetaile();
}

public virtual List<OrderDetails> GetOrderDetails(ir {
    IDictionary outParams = Query<OrderDetails>(ordereturn outParams["orderDetailRowMapper"] as List }
}
```

You can find ready to run code demonstrating the StoredProcedure class in the example 'Data Access' that is part of the Spring.NET distribution.

Chapter 21. Object Relational Mapping (ORM) data access

21.1. Introduction

The Spring Framework provides integration with *NHibernate* in terms of resource management, DAO implementation support, and transaction strategies. For example for NHibernate, there is first-class support with lots of IoC convenience features, addressing many typical NHibernate integration issues. All of these support packages for O/R (Object Relational) mappers comply with Spring's generic transaction and DAO exception hierarchies. There are usually two integration styles: either using Spring's DAO 'templates' or coding DAOs against the 'plain' NHibernate APIs. In both cases, DAOs can be configured through Dependency Injection and participate in Spring's resource and transaction management.

You can use Spring's support for NHibernate without needing to use Spring IoC or transaction management functionality. The NHibernate support classes can be used in typical 3rd party library style. However, usage inside a Spring IoC container does provide additional benefits in terms of ease of configuration and deployment; as such, most examples in this section show configuration inside a Spring container.

Some of the benefits of using the Spring Framework to create your ORM DAOs include:

- *Ease of testing*. Spring's IoC approach makes it easy to swap the implementations and config locations of Hibernate SessionFactory instances, ADO.NET DbProvider instances, transaction managers, and mapper object implementations (if needed). This makes it much easier to isolate and test each piece of persistence-related code in isolation.
- Common data access exceptions. Spring can wrap exceptions from your O/R mapping tool of choice, converting them from proprietary exceptions to a common runtime DataAccessException hierarchy. You can still trap and handle exceptions anywhere you need to. Remember that ADO.NET exceptions (including DB specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with ADO.NET within a consistent programming model.
- General resource management. Spring application contexts can handle the location and configuration of Hibernate ISessionFactory instances, ADO.NET DbProvider instances and other related

resources. This makes these values easy to manage and change. Spring offers efficient, easy and safe handling of persistence resources. For example: related code using NHibernate generally needs to use the same NHibernate Session for efficiency and proper transaction handling. Spring makes it easy to transparently create and bind a Session to the current thread, either by using an explicit 'template' wrapper class at the code level or by exposing a current Session through the Hibernate SessionFactory (for DAOs based on plain Hibernate 1.2 API). Thus Spring solves many of the issues that repeatedly arise from typical NHibernate usage, for any transaction environment (local or distributed).

Integrated transaction management. Spring allows you to wrap your O/R mapping code with either a declarative, AOP style method interceptor, or an explicit 'template' wrapper class at the code level. In either case, transaction semantics are handled for you, and proper transaction handling (rollback, etc) in case of exceptions is taken care of. As discussed below, you also get the benefit of being able to use and swap various transaction managers, without your Hibernate/ADO.NET related code being affected: for example, between local transactions and distributed, with the same full services (such as declarative transactions) available in both scenarios. As an additional benefit, ADO.NET-related code can fully integrate transactionally with the code you use to do O/R mapping. This is useful for data access that's not suitable for O/R mapping which still needs to share common transactions with ORM operations.

The NHibernate Northwind example in the Spring distribution shows a NHibernate implementation of a persistence-technology agnostic DAO interfaces. (In the upcoming RC1 release the SpringAir example will demonstrate an ADO.NET and NHibernate based implementation of technology agnostic DAO interfaces.) The NHibernate Northwind example serves as a working sample application that illustrates the use of NHibernate in a Spring web application. It also leverages declarative transaction demarcation with different transaction strategies.

Both NHibernate 1.0 and NHibernate 1.2 are supported. Differences relate to the use of generics and new features such as contextual sessions. For information on the latter, refer to the section <u>Implementing DAOs based on the plain NHibernate API</u>. The NHibernate 1.0 support is in the assembly Spring.Data.NHibernate and the 1.2 support is in the assembly Spring.Data.NHibernate12

At the moment the only ORM supported in NHibernate, but others can be integrated with Spring (in as much as makes sense) to offer the same value proposition.

21.2. NHibernate

We will start with a coverage of NHibernate in a Spring environment, using it to demonstrate the approach that Spring takes towards integrating O/R mappers. This section will cover many issues in detail and show different variations of DAO implementations and transaction demarcations. Most of these patterns can be directly translated to all other supported O/R mapping tools.

The following discussion focuses on Hibernate 1.0.4, the major differences with NHibernate 1.2 being the ability to participate in Spring transaction/session management via the normal NHibernate API instead of the 'template' approach. Spring supports both NHibernate 1.0 and NHibernate 1.2 via separate .dlls with the same internal namespace.

21.2.1. Resource management

Typical business applications are often cluttered with repetitive resource management code. Many projects try to invent their own solutions for this issue, sometimes sacrificing proper handling of failures for programming convenience. Spring advocates strikingly simple solutions for proper resource handling, namely IoC via templating; for example infrastructure classes with callback interfaces, or applying AOP interceptors. The infrastructure cares for proper resource handling, and for appropriate conversion of specific API exceptions to a common infrastructure exception hierarchy. Spring introduces a DAO exception hierarchy, applicable to any data access strategy. For direct ADO.NET, the AdoTemplate class mentioned in a previous section cares for connection handling, and for proper conversion of ADO.NET data access exceptions (not even singly rooted in .NET 1.1) to Spring's DataAccessException hierarchy, including translation of database-specific SQL error codes to meaningful exception classes. It supports both distributed and local transactions, via respective Spring transaction managers.

also offers of **Spring** Hibernate support, consisting a HibernateTemplate AdoTemplate, analogous to a HibernateInterceptor, and a Hibernate transaction manager. The major goal is to allow for clear application layering, with any data access and transaction technology, and for loose coupling of application objects. No more business service dependencies on the data access or transaction strategy, no more hard-coded resource lookups, no more hard-to-replace singletons, no more custom service registries. One simple and consistent approach to wiring up application objects, keeping them as reusable as possible. All the individual data access features are usable on their own but integrate nicely with Spring's application context concept, providing XML-based configuration and cross-referencing of plain object instances that don't need to be Spring-aware. In a typical Spring application, many important objects are plain .NET objects: data access templates, data access objects (that use the templates), transaction managers, business services (that use the data access objects and transaction managers), ASP.NET web pages (that use the business services), and so on.

21.2.2. Transaction Management

While NHibernate offers an API for transaction management you will quite likely find the benefits of using Spring's generic transaction management features to be more compelling to use, typically for use of a declarative programming model for transaction demarcation and easily mixing ADO.NET and NHibernate operations within a single transaction. See the chapter on transaction management for more information on Spring's transaction management features. There are two choices for transaction management strategies, one based on the NHibernate API and the other the .NET 2.0 TransactionScope API.

encapsulated The first strategy is in the class Spring.Data.NHibernate.HibernateTransactionMana in both the Spring.Data.NHibernate namespace. This strategy is preferred when you are using a single database. ADO.NET operations can also participate in the same transaction, either by using AdoTemplate or by retrieving the ADO.NET connection/transaction object pair stored in thread local storage when the transaction begins. Refer to the documentation of Spring's ADO.NET framework for more information on retrieving and connection/transaction pair without using AdoTemplate. You can use the HibernateTransactionManager and associated classes such as SessionFactory, HibernateTemplate directly as you would any third party API, however they are most commonly used through Spring's XML configuration file to gain the benefits of easy configuration for a particular runtime environment and as the basis for the configuration of a data access layer also configured using XML. An **XML** declaration fragment the of showing HibernateTransactionManager is shown below.

The important property of **HibernateTransactionManager** are the references to the DbProvider and the Hibernate ISessionFactory. For more information on the DbProvider, refer to the chapter <u>DbProvider</u> and the following section on SessionFactory set up.

The second strategy is to use the class Sping.Data.TxScopeTransactionManager that uses .NET 2.0 System.Transaction namespace and its corresponding TransactionScope API. This is preferred when you are using multiple transactional resources, such as multiple databases.

Both strategies associate one Hibernate Session for the scope of the transaction (scope in the general demarcation sense, not System.Transaction sense). If there is no transaction then a new Session will be opened for each operation. The exception to this rule is when using the <code>OpenSessionInViewModule</code> in a web application in single session mode (see Section 21.2.10. "Web Session Management"). In this case the session will be created on the start of the web request and closed on the end of the request. Note that the session's flush mode will be set to <code>FlushMode.NEVER</code> at the start of the request. If a non-readonly transaction is performed, then during the scope of that transaction processing the flush mode will be changed to AUTO, and then set back to NEVER at the end of the transaction scope so that any changes to objects associated with the session during rendering will not be persisted back to the database when the session is closed at the end of the web request.

21.2.3. SessionFactory set up in a Spring container

To avoid tying application objects to hard-coded resource lookups, Spring allows you to define resources like a <code>DbProvider</code> or a Hibernate <code>SessionFactory</code> as objects in an application context. Application objects that need to access resources just receive references to such pre-defined

instances via object references (the DAO definition in the next section illustrates this). The following excerpt from an XML application context definition shows how to set up Spring's ADO.NET DbProvider and a Hibernate SessionFactory on top of it:

```
<objects xmlns="http://www.springframework.net"</pre>
         xmlns:db="http://www.springframework.net/database">
  <!-- Property placeholder configurer for database settings
  <object type="Spring.Objects.Factory.Config.PropertyPlace"</pre>
    cproperty name="ConfigSections" value="databaseSettings"
  </object>
  <!-- Database and NHibernate Configuration -->
  <db:provider id="DbProvider"</pre>
                    provider="SqlServer-1.1"
                    connectionString="Integrated Security=fal
  <object id="MySessionFactory" type="Spring.Data.NHibernate</pre>
    property name="DbProvider" ref="DbProvider"/>
    property name="MappingAssemblies">
      st>
        <value>Spring.Northwind.Dao.NHibernate</value>
      </list>
    </property>
    property name="HibernateProperties">
      <dictionary>
        <entry key="hibernate.connection.provider"</pre>
               value="NHibernate.Connection.DriverConnection
        <entry key="hibernate.dialect"</pre>
               value="NHibernate.Dialect.MsSql2000Dialect"/>
        <entry key="hibernate.connection.driver_class"</pre>
               value="NHibernate.Driver.SqlClientDriver"/>
```

```
</dictionary>
</property>
</object>
</objects>
```

Many of the properties on LocalSessionFactoryObject are those commonly configure, for example the MappingAssemblies specifies a list of assemblies to seach for hibernate mapping files. The property HibernateProperies are the familiar NHibernate properties used to set typical options such as dialect and driver class. The location of NHibernate mapping information can also be specified using Spring's <u>IResource abstraction</u> via the property MappingResources. The IResource abstraction supports opening an input stream from assemblies, file system, and http(s) based on a Uri syntax. You can also leverage the extensibility of IResource and thereby allow NHibernate to obtain its configuration information from locations such as a database or LDAP. For other properties you can configure them as you normal using the file hibernate.cfg.xml and refer to it via the property ConfigFileNames. This property is a string array so multiple configuration files are supported.

There are other properties in LocalSessionFactoryObject that relate to the integration of Spring with NHibernate. The property ExposeTransactionAwareSessionFactory is discussed below and allows you to use Spring's declarative transaction demarcation functionality with the standard NHibernate API (as compared to using HibernateTemplate).

The property **DbProvider** is used to infer two NHibernate configurations options.

- Infer the connection string, typically done via the hibernate property "hibernate.connection_string".
- Delegate to the **DbProvider** itself as the NHibernate connection provider instead of listing it via property hibernate.connection.provider via **HibernateProperties**.

If you specify both the property hibernate.connection.provider and DbProvider

(as shown above) the configuration of the property hibernate.connection.provider is used and a warning level message is logged. If you use Spring's <code>DbProvider</code> as the NHibernate connection provider then you can take advantage of <code>IDbProvider</code> implementations that will let you change the connection string at runtime such as <code>UserCredentialsDbProvider</code> and <code>MultiDelegatingDbProvider</code>.



Note

<u>UserCredentialsDbProvider</u> and <u>MultiDelegatingDbProvider</u> only change the connection string at runtime based on values in thread local storage and do not clear out the Hibernate cache that is unique to each **ISessionFactory** instance. As such, they are only useful for selecting at runtime a single database instance. Cleaning up an existing session factory when switching to a new database is left to user code. Creating a new session factory per connection string (assuming the same mapping files can be used across all databases connections) is not currently supported. To support this functionality, you can subclass LocalSessionFactoryObject and override the method ISessionFactory NewSessionFactory(Configuration config) so that it returns an implementation of ISessionFactory that selects among multiple instances based on values in thread local storage, much like the implementation of MultiDelegatingDbProvider.

21.2.4. The HibernateTemplate

The basic programming model for templating looks as follows for methods that can be part of any custom data access object or business service. There are no restrictions on the implementation of the surrounding object at all, it just needs to provide a Hibernate SessionFactory. It can get the latter from anywhere, but preferably as an object reference from a Spring IoC container - via a simple SessionFactory property setter. The following snippets show a DAO definition in a Spring container, referencing the above defined SessionFactory, and an example for a DAO method implementation.

```
public class HibernateCustomerDao : ICustomerDao {
   private HibernateTemplate hibernateTemplate;
   public ISessionFactory SessionFactory
   {
      set { hibernateTemplate = new HibernateTemplate(value)}
   }
   public Customer SaveOrUpdate(Customer customer)
   {
      hibernateTemplate.SaveOrUpdate(customer);
      return customer;
   }
}
```

The HibernateTemplate class provides many methods that mirror the methods exposed on the Hibernate Session interface, in addition to a number of convenience methods such as the one shown above. If you need access to the Session to invoke methods that are not exposed on the HibernateTemplate, you can always drop down to a callback-based approach like so.

```
public class HibernateCustomerDao : ICustomerDao {
  private HibernateTemplate hibernateTemplate;
  public ISessionFactory SessionFactory
  {
    set { hibernateTemplate = new HibernateTemplate(value) }
    public Customer SaveOrUpdate(Customer customer) {
```

Using the anonymous delegate is particularly convenient when you would otherwise be passing various method parameter calls to the interface based version of this callback. Furthermore, when using generics, you can avoid the typecast and write code like the following

where code is a variable in the surrounding block, accessible inside the anonymous delegate implementation.

A callback implementation effectively can be used for any Hibernate data access. HibernateTemplate will ensure that Session instances are properly opened and closed, and automatically participate in transactions. The template instances are thread-safe and reusable, they can thus be kept as instance variables of the surrounding class. For simple single step actions like a single Find, Load, SaveOrUpdate, or Delete call, HibernateTemplate offers alternative convenience methods that can replace such one line callback provides implementations. Furthermore. Spring a convenient HibernateDaoSupport provides base class that SessionFactory property for receiving a SessionFactory and for use by subclasses. In combination, this allows for very simple DAO implementations for typical requirements:

```
public class HibernateCustomerDao : HibernateDaoSupport, ICu
{
    public Customer SaveOrUpdate(Customer customer)
    {
        HibernateTemplate.SaveOrUpdate(customer);
        return customer;
    }
}
```

21.2.5. Implementing Spring-based DAOs without callbacks

As an alternative to using Spring's HibernateTemplate to implement DAOs, data access code can also be written in a more traditional fashion, without wrapping the Hibernate access code in a callback, while still respecting and participating in Spring's generic DataAccessException hierarchy. The HibernateDaoSupport base class offers methods to access the current transactional Session and to convert exceptions in such a scenario; similar methods are also available static helpers as SessionFactoryUtils class. Note that such code will usually pass 'false' the DoGetSession(..) of as the value 'allowCreate' argument, to enforce running within a transaction (which avoids the need to close the returned **Session**, as its lifecycle is managed by the transaction). Asking for the

```
public class HibernateProductDao : HibernateDaoSupport, IPro
    public Customer SaveOrUpdate(Customer customer)
    {
        ISession session = DoGetSession(false);
        session.SaveOrUpdate(customer);
        return customer;
     }
}
```

This code will *not* translate the Hibernate exception to a generic **DataAccessException**.

21.2.6. Implementing DAOs based on plain Hibernate 1.2/2.0 API

Hibernate 1.2 introduced a feature called "contextual Sessions", where Hibernate itself manages one current <code>ISession</code> per transaction. This is roughly equivalent to Spring's synchronization of one Hibernate <code>Session</code> per transaction. A corresponding DAO implementation looks like as follows, based on the plain Hibernate API:

```
public class ProductDaoImpl implements IProductDao {
    private SessionFactory sessionFactory;
    public ISessionFactory SessionFactory
        get { return sessionFactory; }
        set { sessionFactory = value; }
    }
    public IList<Product> LoadProductsByCategory(String cate
        return SessionFactory.GetCurrentSession()
                .CreateQuery("from test.Product product wher
                .SetParameter(0, category)
                .List<Product>();
    }
}
public class HibernateCustomerDao : ICustomerDao {
  private ISessionFactory sessionFactory;
  public ISessionFactory SessionFactory
      set { sessionFactory = value; }
  }
  public Customer SaveOrUpdate(Customer customer)
      sessionFactory.GetCurrentSession().SaveOrUpdate(custom
      return customer;
```

```
}
}
```

The above DAO follows the Dependency Injection pattern: it fits nicely into a Spring IoC container, just like it would if coded against Spring's HibernateTemplate. Of course, such a DAO can also be set up in plain C# (for example, in unit tests): simply instantiate it and call SessionFactory property with the desired factory reference. As a Spring object definition, it would look as follows:

The SessionFactory configuration to support this programming model can be done two ways, both via configuration of Spring's LocalSessionFactoryObject. You can enable the use of Spring's implementation of the NHibernate extension interface. ICurrentSessionContext, by setting the property $'Expose Transaction Aware Session Factory'\ to\ true\ on\ Local Session Factory Object.$ This is a short-cut for setting the **NHibernate** iust property current_session_context_class with the name of the implementation class to use.

The first way is shown below

Which is simply a shortcut for the following configuration

```
<object id="sessionFactory" type="Spring.Data.NHibernate.Loc</pre>
```

The main advantage of this DAO style is that it depends on the Hibernate API only; no import of any Spring class is required. This is of course appealing from a non-invasiveness perspective, and will no doubt feel more natural to Hibernate developers.

21.2.6.1. Exception Translation

However, the DAO implemenation shown throws plain as HibernateException which means that callers can only treat exceptions as generally fatal - unless they want to depend on Hibernate's own exception hierarchy. Catching specific causes such as an optimistic locking failure is not possible without tying the caller to the implementation strategy. This trade off might be acceptable to applications that are strongly Hibernatebased and/or do not need any special exception treatment. As an alternative you can use Spring's exception translation advice to convert the NHibernate exception to Spring's DataAccessException hierarchy.

Spring offers a solution allowing exception translation to be applied transparently through the [Repository] attribute:

```
[Repository]
public class HibernateCustomerDao : ICustomerDao {
    // class body here
}
```

and register an exception translation post processor.

```
<objects>
  <!-- configure session factory (omittied for brevity) -->
  <!-- Exception translation object post processor -->
  <object type="Spring.Dao.Attributes.PersistenceExceptionTr</pre>
  <!-- Same DAO configuration as before -->
  <object id="CustomerDao" type="Spring.Northwind.Dao.NHiber</pre>
    property name="sessionFactory" ref="MySessionFactory"/>
  </object>
</objects>
```

The postprocessor will automatically look for all exception translators (implementations of the IPersistenceExceptionTranslator interface) and advise all object marked with the [Repository] attribute so that the discovered translators can intercept and apply the appropriate translation on the thrown exceptions. Spring's LocalSessionFactory object $implements \ the \ \textbf{IPersistenceExceptionTranslator} \ interface$ and performs the same exception translation as was done when using HibernateTemplate.

The [Repository] attribute is defined in the Spring.Data assembly, however it is used as a 'marker' attribute, and you can provide your own if you would like to avoid coupling your DAO implementation to a Spring attribute. This done setting PersistenceExceptionTranslationPostProcessor's property RepositoryAttributeType to your own attribute type.



In summary: DAOs can be implemented based on the plain Hibernate 1.2/2.0 API, while still being able to participate in Spring-managed transactions and exception translation.

21.2.7. Programmatic transaction demarcation

Transactions can be demarcated in a higher level of the application, on top of such lower-level data access services spanning any number of operations. There are no restrictions on the implementation of the surrounding business service here as well, it just needs a Spring PlatformTransactionManager. Again, the latter can come from anywhere, but preferably as an object reference via a TransactionManager property - just like the productDAO should be set via a SetProductDao(..) method. The following snippets show a transaction manager and a business service definition in a Spring application context, and an example for a business method implementation.

```
public class FulfillmentService : IFulfillmentService

private TransactionTemplate transactionTemplate;

private IProductDao productDao;

private ICustomerDao customerDao;
```

```
private IOrderDao orderDao;
    private IShippingService shippingService;
    public TransactionManager TransactionManager
       set { transactionTemplate = new TransactionTemplate(\sqrt{
    public void ProcessCustomer(string customerId)
      tt.Execute(delegate(ITransactionStatus status)
          {
            //Find all orders for customer
            Customer customer = CustomerDao.FindById(custome
            foreach (Order order in customer.Orders)
                //Validate Order
                Validate(order);
                //Ship with external shipping service
                ShippingService.ShipOrder(order);
                //Update shipping date
                order.ShippedDate = DateTime.Now;
                //Update shipment date
                OrderDao.SaveOrUpdate(order);
                //Other operations...Decrease product quanti
            return null;
          });
    }
}
```

21.2.8. Declarative transaction demarcation

Alternatively, one can use Spring's declarative transaction support, which essentially enables you to replace explicit transaction demarcation API calls in your C# code with an AOP transaction interceptor configured in a Spring

container. You can either externalize the transaction semantics (like propagation behavior and isolation level) in a configuration file or use the Transaction attribute on the service method to set the transaction semantics.

An example showing attribute driven transaction is shown below

Note that with the new transaction namespace, you can replace the importing of DeclarativeServicesAttributeDriven.xml with the following single line, <tx:attribute-driven/> that more clearly expresses the intent as compared to the contents of DeclarativeServicesAttributeDriven.xml.

The placement of the transaction attribute in the service layer method is shown below.

```
//Ship with external shipping service
ShippingService.ShipOrder(order);

//Update shipping date
order.ShippedDate = DateTime.Now;

//Update shipment date
OrderDao.SaveOrUpdate(order);

//Other operations...Decrease product quanti
}

}
```

If you prefer to not use attribute to demarcate your transaction boundaries, you can import a configuration file with the following XML instead of using <tx:attribute-driven/>

Refer to the documentation on Spring Transaction management for configuration of other features, such as rollback rules.

21.2.9. Transaction management strategies

Both TransactionTemplate and TransactionInterceptor (not yet seen explicitly in above configuration, TransactionProxyFactoryObject uses a TransactionInterceptor, you would have to specify it explicitly if you were using an ordinary ProxyFactoryObject.) delegate the actual transaction handling

to a PlatformTransactionManager instance, which can be a HibernateTransactionManager (for single a Hibernate SessionFactory, using a ThreadLocal Session under the hood) or a TxScopeTransactionManager (delegating to MS-DTC for distributed transaction) for Hibernate applications. You could even use a PlatformTransactionManager custom implementation. from native Hibernate transaction switching management TxScopeTransactionManager, such as when facing distributed transaction requirements for certain deployments of your application, is just a matter of configuration. Simply replace the Hibernate transaction manager with Spring's TxScopeTransactionManager implementation. Both transaction demarcation and data access code will work without changes, as they just use the generic transaction management APIs.

For distributed transactions across multiple Hibernate session factories, simply combine TxScopeTransactionManager as a transaction strategy with multiple LocalSessionFactoryObject definitions. Each of your DAOs then gets one specific SessionFactory reference passed into it's respective object property.

```
TO BE DONE
```

HibernateTransactionManager can export the ADO.NET Transaction used by Hibernate to plain ADO.NET access code, for a specific DbProvider. (matching connection string). This allows for high-level transaction demarcation with mixed Hibernate/ADO.NET data access!

21.2.10. Web Session Management

The open session in view pattern keeps the hibernate session open during page rendering so lazily loaded hibernate objects can be displayed. You configure its use by adding an additional custom HTTP module declaration as shown below

```
<system.web>
  <httpModules>
     <add name="OpenSessionInView" type="Spring.Data.NHiber
  </httpModules>
```

```
...
</system.web>
```

You can configure which SessionFactory the OpenSessionInViewModule will use by setting 'global' application key-value pairs as shown below. (this will change in future releases)

```
<appSettings>
    <add key="Spring.Data.NHibernate.Support.OpenSessionInVi
</appSettings>
```

The default behavior of the module is that a single session is currently used for the life of the request. Refer to the earlier section on Transaction Management in this chapter for more information on how sessions are managed in the OpenSessionInViewModule. You can also configure in the application setting EntityInterceptorObjectName the using the key Spring.Data.NHibernate.Support.OpenSessionInView SingleSession mode is used via Spring.Data.NHibernate.Support.OpenSessionInVie If SingleSession is set to false, referred to as 'deferred close mode', then each transaction scope will use a new Session and kept open until the end of the web request. This has the drawback that the first level cache is not reused across transactions and that objects are required to be unique across all sessions. Problems can arise if the same object is associated with more than one hibernate session.



Important

By default, OSIV applies FlushMode. NEVER on every session it creates. This is because if OSIV flushed pending changes during "EndRequest" and an error occurs, all response has already been sent to the client. There would be no way of telling the client about the error.

By default this means you MUST explicitly demarcate transaction boundaries around non-readonly statements when using OSIV. For configuring transactions see Section 21.2.8, "Declarative transaction demarcation" or the Spring.Data.NHibernate.Northwind example application.

21.2.11. Session Scope

The class Spring.Data.NHibernate.Support.SessionScope allows for you to use a single NHibernate session across multiple transactions. The usage is shown below

```
using (new SessionScope())
{
    ... do multiple operations with a single session, possib
}
```

At the end of the using block the session is automatically closed. All transactions within the scope use the same session, if you are using Spring's HibernateTemplate or using Spring's implementation of NHibernate 1.2's ICurrentSessionContext interface. See other sections in this chapter for further information on those usage scenarios.

Spring Framework



Part III. The Web

This part of the reference documentation covers the Spring Framework's support for the presentation tier, specifically web-based presentation tiers.

- Chapter 22, Spring.NET Web Framework
- Chapter 23, ASP.NET AJAX

Chapter 22. Spring.NET Web Framework

22.1. Introduction

Spring.NET's web application framework aims to increase your productivity writing ASP.NET WebForms applications. It offers a unique value proposition to creating ASP.NET applications not found in other .NET web frameworks.

The goal of the framework is to make it easy to write 'thin and clean' web applications. By thin, what is meant is that the WebForm's responsibility is to act as adapter between the HTML based world of the web and the oo world of your application. The application layer your web form communicates with is where the business logic resides, not in the web tier. By 'clean' what is meant that the web framework should have a good separation of concerns, leading ideally to an event-handler that does not contain any reference to UI elements. This makes it possible to test your event handler code in integration style tests. Last but not least, Spring's web framework reduces the incidental complexity of common tasks in the web tier, for example the conversion of HTML control data to objects and then vice-versa after the request has been processed by the application layer.

Highlights of Spring's Web framework are

- Dependency Injection for all ASP.NET artifacts. This includes pages and user controls but also modules, providers and HTTP handlers. Your pages, controls, etc., do not have any Spring dependencies in order to be configured via dependency injection.
- Bi-directional data binding. This allows you to declaratively define the data that will be marshaled out of your html/user controls and into a data model that in turn is generally submitted to the application layer. After the data model is updated in the application layer, those changes are automatically reflected in the html/user controls on post back. This removes large amounts of tedious, error prone boilerplate code.
- Web object scopes. Object definitions can be defined at the application, session or request scope. This makes it easy to inject, say a session scoped shopping cart, into your page without having to do any lower level programming.
- <u>Data Model Management</u>. While ASP.NET managed the view state of your form, it does not offer facilities to manage the data model that you build up to submit to the application layer. Spring provides a mechanism similar to

view state to help manage your data model.

- UI agnostic validation framework. Declaratively define complex validation rules, for example that take into account complex relationships in your data model. Error controls are provided to easily render validation failure. This allows you to centralize your validation logic and also reuse it on the server side, for example using parameter validation advice described in the aspect library chapter
- Externalized page navigation through 'result mapping'. Instead of hard coding urls and data to direct where a page should go next, result mappings are externally defined and configured that associate logical names and a URL (+ data). This also allows for the encryption of values that are sent via Response.Redirect.
- Improved <u>localization</u> and master page support Advanced localization features (including image localization) as well as declarative configuration of what mater page to apply to different parts of your web application are easy to perform.

All you know about ASP.NET development still applies, Spring's approach is to 'embrace and extend' the basic ASP.NET programming model so you can be as productive as possible.



Support for ASP.NET MVC is planned for Spring.NET 2.0 and previews of our integration with the MVC framework will be mode available when the final MVC framework ships.

What follows is a more detailed background and motivation of features and most importantly the detailed reference manual for using Spring's web framework. One of the great things about the framework is that it is not an all or nothing solution. If you choose to use only dependency injection and bi-directional data binding, that is just fine. You can incrementally adopt the web framework, addressing problems areas in your current web application with a specific feature. There is no need to go 'whole hog' into using all parts of the framework everywhere in your application.

The Spring.NET distribution ships with a number of Web QuickStarts and a complete reference application, SpringAir. Web QuickStarts are the best way to learn each Spring. Web feature by following simple examples, and the SpringAir reference application has a Spring.Web-enabled frontend which uses many best practices for Spring.NET web applications, so please do refer to it as you are reading this (reference) material (see Chapter 37, SpringAir - Reference Application).

22.2. Background

One of the objections many developers have to the ASP.NET programming model is that it is not a "true MVC" (Model-View-Controller) implementation, because controller-type logic within the page is too tightly coupled to the view. A good example of this are event handlers within the page class, which typically have references to view elements, such as input controls, all over the place. Without getting into academic discussion of what "true MVC" is, and whether it is even appropriate to try to fit form-based technology such as ASP.NET into traditionally request-based MVC pattern when MVP (Model-View-Presenter) or Presentation Model might be more appropriate, we'd like to agree with the critics on the most important point they are making: controller-type logic, such as the code within page event handlers in ASP.NET, should not depend on the view elements.

Having said that, there *are* good things about ASP.NET. Server-side forms and controls make developers significantly more productive and allow us to significantly simplify page markup. They also make cross-browser issues easier to deal with, as each control can make sure that it renders correct markup based on the user's browser. The ability to hook custom logic into the lifecycle of the page, as well as to customize HTTP processing pipeline are also very powerful features. Finally, being able to interact with the strongly typed server-side controls instead of manipulating string-based HTTP request collections, such as Form and QueryString, is a much needed layer of abstraction in web development.

For these reasons, we decided that instead of developing a new, "pure and true MVC" web framework as part of Spring.NET, we should take a more pragmatic approach and extend ASP.NET in such a way that most, if not all of its shortcomings are eliminated. It should be noted that with the introduction of a 'true MVC framework' being added to .NET, with extension points for IoC containers such as Spring, Spring will continue to play a role within a MVC based model once that functionality is available from Microsoft. It is worth noting that Spring Java has a very popular MVC framework and much of that experience and added value can be transliterated to help developers be more productive when using the upcoming ASP.NET MVC support.

Spring. Web also adds support for applying the dependency injection principle to one's ASP.NET Pages and Controls as well as http modules and custom

provider modules. This means that application developers can easily inject service dependencies into web controllers by leveraging the power of the Spring.NET IoC container. See <u>Dependency Injection for ASP.NET Pages</u> for more information.

As we said earlier, event handlers in code-behind classes really should not have to deal with ASP.NET UI controls directly. Such event handlers should rather work with the presentation model of the page, represented either as a hierarchy of domain objects or an ADO.NET <code>DataSet</code>. It is for that reason that the Spring.NET team implemented bidirectional data binding framework to handle the mapping of values to and from the controls on a page to the underlying data model. The data binding framework also transparently takes care of data type conversion and formatting, enabling application developers to work with fully typed data (domain) objects in the event handlers of code-behind files. See Bidirectional Data Binding and Model Management for more information.

The flow of control through an application is another area of concern that is addressed by Spring.NET Web Framework. Typical ASP.NET applications will use Response.Redirect or Server.Transfer calls within Page logic to navigate to an appropriate page after an action is executed. This typically leads to hard-coded target URLs in the Page, which is never a good thing. Result mapping solves this problem by allowing application developers to specify aliases for action results that map to target URLs based on information in an external configuration file that can easily be edited. Under consideration for future releases of Spring.NET is a process management framework, which will take this approach to another level, allowing you to control complex page flows in a very simple way. See Result Mapping for more information.

Standard localization support is also limited in versions of ASP.NET prior to ASP.NET 2.0. Even though Visual Studio 2003 generates a local resource file for each ASP.NET Page and user control, those resources are never used by the ASP.NET infrastructure. This means that application developers have to deal directly with resource managers whenever they need access to localized resources, which in the opinion of the Spring.NET team should not be the case. Spring.NET's Web Framework (hereafter referred to as Spring.Web) adds comprehensive support for localization using both local resource files and global resources that are configured within and for a Spring.NET container. See Localization and Message Sources for more information.

In addition to the aforementioned features that can be considered to be the 'core' features of the Spring. Web framework, Spring. Web also ships with a number of other lesser features that might be useful to a large number of application developers. Some of these additional features include back-ports of ASP.NET 2.0 features that can be used with ASP.NET 1.1, such as Master Page support. See Master Pages in ASP.NET 1.1 for more information.

In order to implement some of the above mentioned features the Spring.NET team had to extend (as in the object-oriented sense) the standard ASP.NET Page and UserControl classes. This means that in order to take advantage of the full feature stack of Spring. Web (most notably bidirectional data binding, localization and result mapping), your code-behind classes will specific Spring.Web base classes such have to extend as Spring.Web.UI.Page; however, some very powerful features such as dependency injection for ASP.NET Pages, Controls, and providers can be leveraged without having to extend Spring. Web-specific base classes. It is worth stating that by taking advantage of some of the more useful features offered by Spring. Web you will be coupling the presentation tier of your application(s) to Spring. Web. The choice of whether or not this is appropriate is, of course, left to you.

22.3. Automatic context loading and hierarchical contexts

22.3.1. Configuration

Unsurprisingly, Spring.Web builds on top of the Spring.NET IoC container, and makes heavy use (internally) of the easy pluggability and standardized configuration afforded by the IoC container. This also means that all of the controllers (ASP.NET Pages) that make up a typical Spring.Web enabled application will be configured using the same standard Spring.NET XML configuration syntax. Spring.Web uses a custom PageHandlerFactory implementation to load and configure a Spring.NET IoC container, which is in turn used to locate an appropriate Page to handle a HTTP request. The WebSupportModule configures miscellaneous Spring infrastructure classes for use in a web environment, for example setting the storage strategy of LogicalThreadContext to be HybridContextStorage.

The instantiation and configuration of the Spring.NET IoC container by the Spring.Web infrastructure is wholly transparent to application developers, who will typically never have to explicitly instantiate and configure an IoC container manually (by for example using the <code>New</code> operator in C#). In order to effect the transparent bootstrapping of the IoC container, the Spring.Web infrastructure requires the insertion of the following configuration snippet into each and every Spring.Web-enabled web application's root <code>Web.config</code> file (the <code>verb</code> and <code>path</code> properties can of course be changed from the values that are shown below):

Please note that this snippet of standard ASP.NET configuration is only required

to be present in the *root* directory of each Spring. Web web application (i.e. in the web.config file present in the top level virtual directory of an ASP.NET web application).

The above XML configuration snippet will direct the ASP.NET infrastructure to use Spring.NET's page factory, which will in turn create instances of the appropriate .aspx Page, (possibly) inject dependencies into said Page (as required), and then forward the handling of the request to said Page.

After the Spring. Web page factory is configured, you will also need to define a root application context by adding a Spring. NET configuration section to that same Web. Config file. The final configuration file should look a little like this (your exact configuration will no doubt vary in particulars)...

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <configSections>
        <sectionGroup name="spring">
          <section name="context" type="Spring.Context.Suppo"</pre>
        </sectionGroup>
    </configSections>
    <spring>
        <context>
            <resource uri="~/Config/CommonObjects.xml"/>
            <resource uri="~/Config/CommonPages.xml"/>
            <!-- TEST CONFIGURATION -->
            <!--
            <resource uri="~/Config/Test/Services.xml"/>
            <resource uri="~/Config/Test/Dao.xml"/>
            -->
            <!-- PRODUCTION CONFIGURATION -->
            <resource uri="~/Config/Production/Services.xml'</pre>
            <resource uri="~/Config/Production/Dao.xml"/>
        </context>
    </spring>
```

There are a few important points that need to be noted with regard to the above configuration:

- 1. You must define a custom configuration section handler for the spring/context element. If you use Spring.NET for many applications on the same web server, it might be easier to move the whole definition of the Spring.NET section group to your machine.config file.
- 2. The configuration custom section handler is of the type Spring.Context.Support.WebContextHandler which will in turn instantiate an IoC container of the type Spring.Context.Support.WebApplicationContext. This will ensure that all of the features provided by Spring. Web are handled properly (such as request and session-scoped object definitions).
- 3. Within the <spring> element you need to define a root context, and resource locations that contain the object definitions that will be used within the web application (such as service or business tier objects) then need to be specified as child elements within the <context> element. Object definition resources can be fully-qualified paths or URLs, or non-qualified, as in the example above. Non-qualified resources will be loaded using the default resource type for the context, which for the WebApplicationContext is the WebResource type.
- 4. Please note that the object definition resources do not all have to be the same resource type (e.g. all file://, all http://, all assembly://, etc). This means that you can load some object definitions from resources embedded directly within application assemblies

(assembly://), while continuing to load other object definitions from web resources that can be more easily edited.

22.3.1.1. Configuration for IIS7

The configuration for IIS7 is shown below

22.3.2. Context Hierarchy

ASP.NET provides a hierarchical configuration mechanism by allowing application developers to override configuration settings specified at a higher level in the web application directory hierarchy with configuration settings specified at the lower level.

For example, a web application's root Web.config file overrides settings from the (lower level) machine.config file. In the same fashion, settings specified within the Web.config file within a subdirectory of a web application will override settings from the root Web.config and so on. Lower level Web.config files can also add settings of their own that were not previously defined anywhere.

Spring.Web leverages this ASP.NET feature to provide support for a context hierarchy. Your lower level Web.config files can be used to add new object definitions or to override existing ones per virtual directory.

What this means to application developers is that one can easily componentize an application by creating a virtual directory per component and creating a custom context for each component that contains the necessary configuration info for that particular context. The configuration for a lower level component will generally contain only those definitions for the pages that the component consists of and (possibly) overrides for some of the definitions from the root context (for example, menus).

Because each such lower level component will usually contain only a few object definitions, application developers are encouraged to embed those object definitions directly into the <code>Web.config</code> for the lower level context instead of relying on an external resource containing object definitions. This is easily accomplished by creating a component <code>Web.config</code> similar to the following one:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="spring">
       <section name="objects" type="Spring.Context.Support.</pre>
    </sectionGroup>
  </configSections>
  <spring>
    <context type="Spring.Context.Support.WebApplicationCont</pre>
        <resource uri="config://spring/objects"/>
    </context>
    <objects xmlns="http://www.springframework.net">
        <object type="MyPage.aspx" parent="basePage">
              roperty name="MyRootService" ref="myService"
              property name="MyLocalService" ref="myService"
              property name="Results">
                 <!-- ... -->
              </property>
        </object>
        <object id="myServiceDefinedLocally" type="MyCompany</pre>
    </objects>
  </spring>
</configuration>
```

The <context/> element seen above (contained within the <spring/> element) simply tells the Spring.NET infrastructure code to load (its) object definitions from the Spring/Objects section of the configuration file.

You can (and should) avoid the need to specify <configSections/> element by moving the configuration handler definition for the <Objects> element to a higher level (root) Web.config file, or even to the level of the machine.config file if Spring.NET is to be used for multiple applications on the same server.

A very important point to be aware of is that this component-level context can reference definitions from its parent context(s). Basically, if a referenced object definition is not found in the current context, Spring.NET will search all the ancestor contexts in the context hierarchy until it finds said object definition (or ultimately fails and throws an exception).

22.4. Dependency Injection for ASP.NET Pages

Spring. Web builds on top of the feature set and capabilities of ASP.NET; one example of this can be seen the way that Spring. Web has used the code-behind class of the Page mechanism to satisfy the Controller portion of the pattern. In MVC-based MVC architectural (web) applications, **Controller** is typically a thin wrapper around one or more service objects. In the specific case of Spring.Web, the Spring.NET team realized that it was very important that service object dependencies be easily injected into Page Controllers. Accordingly, Spring. Web provides first class support for dependency injection in ASP.NET Pages. This allows application developers to inject any required service object dependencies (and indeed any other dependencies) into their Pages using standard Spring.NET configuration instead of having to rely on custom service locators or manual object lookups in a Spring.NET application context.

Once an application developer has <u>configured</u> the Spring.NET web application context, said developer can easily create object definitions for the pages that compose that web application:

This example contains three definitions:

1. The first definition is an abstract definition for the base page that many other pages in the application will inherit from. In this case, the definition simply specifies which page is to be referenced as the master page, but it

- will typically also configure localization-related dependencies and root folders for images, scripts and CSS stylesheets.
- 2. The second definition defines a login page that neither inherits from the base page nor references the master page. What it does show is how to inject a service object dependency into a page instance (the authenticationService is defined elsewhere).
- 3. The final definition defines a default application page. In this case it simply inherits from the base page in order to inherit the master page dependency, but apart from that it doesn't need any additional dependency injection configuration.

One thing that slightly differentiates the configuration of ASP.NET pages from the configuration of other .NET classes is in the value passed to the type attribute. As can be seen in the above configuration snippet the type name is actually the path to the .aspx file for the Page, relative to the directory context it is defined in. In the case of the above example, those definitions are in the root context so Login.aspx and Default.aspx also must be in the root of the web application's virtual directory. The master page is defined using an absolute path because it could conceivably be referenced from child contexts that are defined within subdirectories of the web application.

The astute reader may have noticed that the definitions for the Login and Default pages don't specify either of the id and name attributes. This is in marked contrast to typical object definitions in Spring.NET, where the id or name attributes are typically mandatory (although not always, as in the case of inner object definitions). This is actually intentional, because in the case of Spring.Web Page Controller instances one typically wants to use the name of the .aspx file name as the identifier. If an id is not specified, the Spring.Web infrastructure will simply use the name of the .aspx file as the object identifier (minus any leading path information, and minus the file extension too).

Nothing prevents an application developer from specifying an id or name value explicitly; one use case when the explicit naming might be useful is when one wants to expose the same page multiple times using a slightly different configuration (Add / Edit pages for example). If you would like to use abstract object definitions and have your page inherit from them, the use of the name

attribute should be used instead of the id attribute on the abstract object definition.

22.4.1. Injecting Dependencies into Controls

Spring.Web also allows application developers to inject dependencies into controls (both user controls and standard controls) that are contained within a page. This can be accomplished globally for all controls of a particular Type by using the location of the .ascx as the object type identifier. This is similar to injecting into .aspx pages shown above.

```
<object type="~/controls/MyControl.ascx" abstract="true">
    <!-- inject dependencies here... -->
</object>
```

In either case, be sure to mark the object definition as abstract (by adding abstract="true" to the attribute list of the <object/> element).

22.4.2. Injecting dependencies into custom HTTP modules

You can perform dependency injection on custom HTTP modules through the use of the class Spring.Context.Support.HttpApplicationConfigure You register your custom HTTP module as you would normally, for example a module of the type HtmlCommentAppenderModule, taken from the Web Quickstart, appends additional comments into the http response. It is registered as shown below

```
<httpModules>
    <add name="HtmlCommentAppender" type="HtmlCommentAppender" </httpModules>
```

To configure this module, naming conventions are used to identify the module name with configuration instructions in the Spring configuration file. The ModuleTemplates property of HttpApplicationConfigurer is a dictionary that takes as a key the name of the HTTP module, HtmlCommentAppender, and as a value the configuration instructions as you would normally use for configuring an object with Spring. An example is shown below. HttpApplicationConfigurer'

ModuleTemplates property.

You can see this example in action in the Web Quickstart.

22.4.3. Injecting dependencies into HTTP handlers and handler factories

You can perform dependency injection on IHttpHandlers and IHttpHandlerFactory. This effectively allows for a fully Springmanaged httpHandlers configuration section. To configure an IHttpHandler or IHttpHandlerFactory you should register Spring's MappingHandlerFactory with a specific path or wildcard string (i.e. *.aspx) using the standard configuration of an httpHandler in web.config. This is shown below

The specialization of which specific handler is mapped to the path is done by configuration of Spring's

MappingHandlerFactoryConfigurer class.

MappingHandlerFactoryConfigurer is configured by specifying a dictionary of key value paris, the key value is a regular expression that will match the request URL and the value is an instance of an IHttpHandler or IHttpHandlerFactory configured via dependency injection.

the configuration of MappingHandlerFactoryConfigurer is shown below

```
<objects xmlns="http://www.springframework.net">
    <!-- configures the global GenericHandlerFactory instand
    <object name="mappingHandlerFactoryConfigurer" type="Spr</pre>
        property name="HandlerMap">
            <dictionary>
                 <!-- map any request ending with *.whatever
                <entry key="\.whatever$" value="myCustomHand</pre>
                <entry key="\.ashx$" value="standardHandlerF</pre>
            </dictionary>
        </property>
    </object>
    <object name="standardHandlerFactory" type="Spring.Web.$</pre>
    <!-- defines a standard singleton that will handle *.wha
    <object name="myCustomHandler" type="MyCustomHttpHandler</pre>
        property name="MessageText" value="This text is inj
    </object>
    <!--
        used for configuring ~/DemoHandler.ashx custom handl
        note, that this is an abstract definition because 't
    -->
    <object name="DemoHandler.ashx">
        property name="OutputText">
            <value>This text is injected via Spring</value>
        </property>
    </object>
</objects>
```

DefaultHandlerFactory will use the .NET System.Web.UI.SimpleHandlerFactory to create handler instaces and will configure each instances used an object definition whose name matches the The url's filename. abstract obiect definition of request DemoHandler.ashx is an example of this approach. You may also configure standard classes that implment the IHttpHandler interface as demonstrated in the example above for the class MyCustomHttpHandler.

Please refer to the Web Quickstart application too see this in action.

22.4.4. Injecting dependencies into custom providers

Custom providers can be configured with Spring. The approach to configuration is a family of adapters that correspond 1-to-1 with the standard ASP.NET providers that are registered using the standard ASP.NET mechanism. The adapters inherit from their correspondingly named provider class in the BCL.

- MembershipProviderAdapter
- ProfileProviderAdapter
- RoleProviderAdapter
- SiteMapProviderAdapter

Here is an example of how to register the adapter for membership providers.

The name of the provider must match the name of the object in the spring configuration that will serve as the actual provider implementation. For convenience there are configurable versions of the providers found in ASP.NET so that you can use the full functionality of spring to configure these standard provider implementations, for example using property place holders, etc. These are

• ConfigurableActiveDirectoryMembershipProvider

- ConfigurableSqlMembershipProvider
- ConfigurableSqlProfileProvider
- ConfigurableSqlRoleProvider
- ConfigurableXmlSiteMapProvider

Here is an example configuration taken from the Web Quickstart that simply sets the description property and connection string.

Your own custom providers of course will contain additional configuration specific to your implementation.

22.4.5. Customizing control dependency injection

There might be situations where it is necessary to customize Spring.Web's dependency injection processing. In particular when using GridViews, which create a large number of child controls, dependency injection can slow down your page. To overcome this problem, you may tell Spring to handle the dependency injection process yourself by implementing the interface ISupportsWebDependencyInjection as shown below:

```
[C#]
class MyControl : Control, ISupportsWebDependencyInjection
{
  private IApplicationContext _defaultApplicationContext;

  public IApplicationContext DefaultApplicationContext
  {
    get { return _defaultApplicationContext; }
    set { _defaultApplicationContext = value; }
}
```

```
override protected AddedControl( Control control, int inde
{
    // handle DI for children ourselves -
    // defaults to a call to InjectDependenciesRecursive
    WebUtils.InjectDependenciesRecursive( _defaultApplication base.AddedControl( control, index );
}
```

There is a Spring server control, Panel, that provides an easier way to turn of dependency injection for parts of your page. Example use is shown below

```
<spring:Panel runat="server"
    suppressDependencyInjection="true"
    renderContainerTag="false">
    .. put your heavy controls here - they won't be touched be contained.
```

By wrapping the performance sensitive parts of your page within this panel, you can easily turn off DI using the attribute suppressDependencyInjection. By default <spring:Panel/> won't render a container tag (<div>, , etc.). You can modify this behavior by setting the attribute "renderContainerTag" accordingly.

22.5. Web object scopes

Spring.NET web applications support an additional attribute within object definition elements that allows you to control the scope of an object:

```
<object id="myObject" type="MyType, MyAssembly" scope="appli</pre>
```

As you can see, there are three possible values for the scope attribute -- application, session or request. Application scope is the default, and will be used for all objects that don't have scope attribute defined. As its name says, it will result in a single instance of an object being created for the duration of the application, so it works exactly like the standard singleton objects in non-web applications. Session scope allows you to define objects in such a way that an instance is created for each HttpSession. This is the ideal scope for objects that you want bound to a single user such as user profile, shopping cart, etc. Request scope will result in a creation of an instance per HTTP request.

Unlike with prototype objects, calls to IApplicationContext.GetObject will return the same instance of the request-scoped object during a single HTTP request. This allows you, for example, to inject the same request-scoped object into multiple pages and then use server-side transfer to move from one page to another. As all the pages are executed within the single HTTP request in this case, they will share the same instance of the injected object.

One thing to keep in mind is that objects can only reference other objects that are in the same or broader scope. This means that application-scoped objects can only reference other application-scoped, session-scoped objects can reference both session and application-scoped objects, and finally, request-scoped objects can reference other request, session or application-scoped objects. Also, prototype objects (and that includes all ASP.NET web pages defined within Spring.NET context) can reference singleton objects from any scope, as well as other prototype objects.

22.6. Master Pages in ASP.NET 1.1

Support for ASP.NET 1.1 master pages in Spring.Web is very similar to the support for master pages in ASP.NET 2.0.

The idea is that a web developer can define a layout template for the site as a master page and specify content place holders that other pages can then reference and populate. A sample master page (MasterLayout.ascx) could look like this:

```
<%@ Control language="c#" Codebehind="MasterLayout.ascx.cs"</pre>
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Cor</pre>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN</pre>
<html>
   <head>
       <title>Master Page</title>
       <spring:ContentPlaceHolder id="head" runat="server"/</pre>
   </head>
   <body>
      <form runat="server">
          <spring:ContentPlaceHolder id="title"</pre>
                        <!-- default title content -->
                    </spring:ContentPlaceHolder>
                 <tr>
                 <spring:ContentPlaceHolder id="lefts"</pre>
                        <!-- default left side content
                    </spring:ContentPlaceHolder>
                 <spring:ContentPlaceHolder id="main"</pre>
                        <!-- default main area content
                    </spring:ContentPlaceHolder>
```

```
</form>
</body>
</html>
```

As you can see from the above code, the master page defines the overall layout for the page, in addition to four content placeholders that other pages can override. The master page can also include default content within the placeholder that will be displayed if a derived page does not override the placeholder.

A page (Child.aspx) that uses this master page might look like this:

The <spring:Content/> control in the above example uses the contentPlaceholderId attribute (property) to specify exactly which placeholder from the master page is to be overridden. Because this particular page does not define content elements for the head and title place holders, they will be displayed using the default content supplied in the master page.

Both the ContentPlaceHolder and Content controls can contain any valid ASP.NET markup: HTML, standard ASP.NET controls, user controls, etc.



VS.NET 2003 issue

Technically, the <html> and <body> tags from the previous example are not strictly necessary because they are already defined in

the master page. However, if these tags are omitted, then Visual Studio 2003 will complain about a schema and intellisense won't work, so it's much easier to work in the HTML view if those tags are included. They will be ignored when the page is rendered.

22.6.1. Linking child pages to their master

The Spring.Web.UI.Page class exposes a property called MasterPageFile, which can be used to specify the master page.

The recommended way to do this is by leveraging the Spring.NET IoC container and creating definitions similar to the following:

This approach allows application developers to change the master page being used for a number of pages within a web application. Of course, the master page can still be overridden on a per context or per page basis by creating a new abstract page definition within a child context, or by specifying the MasterPageFile property directly.

22.7. Bidirectional Data Binding and Model Management

A problem with the existing data binding support in ASP.NET is that it is one-way only. It allows application developers to bind page controls to the data model and display information from said data model, but it doesn't allow for the extraction of values from the controls when the form is submitted. Spring.Web adds such bidirectional data binding to ASP.NET by allowing developers to specify data binding rules for their page, and by automatically evaluating configured data binding rules at the appropriate time in the page's lifecycle.

ASP.NET also doesn't provide any support for model management within the postbacks. Sure, it has a ViewState management, but that takes care of the control state only and not of the state of any presentation model objects these controls might be bound to. In order to manage model within ASP.NET, developers will typically use HTTP Session object to store the model between the postbacks. This results in a decent amount of boilerplate code that can and should be eliminated, which is exactly what Spring.Web does by providing a simple set of model management methods.

Please note that in order to take advantage of the bidirectional data binding and model management support provided by Spring.Web, you will have to couple your presentation layer to Spring.Web; this is because features requires you to extend a Spring.Web.UI.Page instead of the usual System.Web.UI.Page class.

Spring.Web data binding is very easy to use. Application developers simply need to override the protected InitializeDataBindings method and configure data binding rules for the page. They also need to override three model management methods: InitializeModel, LoadModel and SaveModel. This is perhaps best illustrated by an example from the SpringAir reference application. First, let's take a look at the page markup:

```
<spring:RadioButtonGroup ID="tripMode" r</pre>
                <asp:RadioButton ID="OneWay" runat="</pre>
                <asp:RadioButton ID="RoundTrip" runa</pre>
             </spring:RadioButtonGroup>
          <tr>
          <asp:Label ID="leavingFrom" runat="serve</pre>
          <asp:DropDownList ID="leavingFromAirport</pre>
          <asp:Label ID="goingTo" runat="server"</pre>
          <asp:DropDownList ID="goingToAirportCode"</pre>
          <tr>
          <asp:Label ID="leavingOn" runat="server"</pre>
          <spring:Calendar ID="departureDate" runa</pre>
          <asp:Label ID="returningOn" runat="serve"</pre>
          <div id="returningOnCalendar">
                <spring:Calendar ID="returnDate" rur</pre>
             </div>
          <tr>
          <br/>
             <asp:Button ID="findFlights" runat="serv</pre>
        </div>
</asp:Content>
```

Ignore for the moment the fact that none of the label controls have text defined, which will be described later when we discuss localization in Spring.NET. What is important for the purposes of our current discussion, is that we have a number of input controls defined: tripMode radio group, leavingFromAirportCode and goingToAirportCode dropdowns, as well as two Spring.NET Calendar controls, departureDate and returnDate.

Next, let's take a look at the model we will be binding this form to:

```
namespace SpringAir.Domain
{
    [Serializable]
    public class Trip
    {
        // fields
        private TripMode mode;
        private TripPoint startingFrom;
        private TripPoint returningFrom;
        // constructors
        public Trip()
        {
            this.mode = TripMode.RoundTrip;
            this.startingFrom = new TripPoint();
            this.returningFrom = new TripPoint();
        }
        public Trip(TripMode mode, TripPoint startingFrom,
        {
            this.mode = mode;
            this.startingFrom = startingFrom;
            this.returningFrom = returningFrom;
        }
        // properties
        public TripMode Mode
            get { return this.mode; }
```

```
set { this.mode = value; }
    }
    public TripPoint StartingFrom
        get { return this.startingFrom; }
        set { this.startingFrom = value; }
    }
    public TripPoint ReturningFrom
        get { return this.returningFrom; }
        set { this.returningFrom = value; }
    }
}
[Serializable]
public class TripPoint
    // fields
    private string airportCode;
    private DateTime date;
    // constructors
    public TripPoint()
    {}
    public TripPoint(string airportCode, DateTime date)
    {
        this.airportCode = airportCode;
        this.date = date;
    }
    // properties
    public string AirportCode
    {
        get { return this.airportCode; }
        set { this.airportCode = value; }
    }
    public DateTime Date
```

```
get { return this.date; }
    set { this.date = value; }
}

[Serializable]
public enum TripMode
{
    OneWay,
    RoundTrip
}
```

As you can see, Trip class uses the TripPoint class to represent departure and return, which are exposed as StartingFrom and ReturningFrom properties. It also uses TripMode enumeration to specify whether the trip is one way or return trip, which is exposed as Mode property.

Finally, let's see the code-behind class that ties everything together:

```
public class TripForm : Spring.Web.UI.Page
    // model
    private Trip trip;
    public Trip Trip
    {
        get { return trip; }
        set { trip = value; }
    }
    // service dependency, injected by Spring IoC container
    private IBookingAgent bookingAgent;
    public IBookingAgent BookingAgent
    {
        set { bookingAgent = value; }
    }
    // model management methods
    protected override void InitializeModel()
```

```
trip = new Trip();
                          trip.Mode = TripMode.RoundTrip;
                          trip.StartingFrom.Date = DateTime.Today;
                          trip.ReturningFrom.Date = DateTime.Today.AddDays(1);
             }
             protected override void LoadModel(object savedModel)
                          trip = (Trip) savedModel;
             }
             protected override object SaveModel()
             {
                          return trip;
             }
             // data binding rules
            protected override void InitializeDataBindings()
                          BindingManager.AddBinding("tripMode.Value", "Trip.Mode.Value", "Trip.M
                          BindingManager.AddBinding("leavingFromAirportCode.Se
                          BindingManager.AddBinding("goingToAirportCode.Select
                          BindingManager.AddBinding("departureDate.SelectedDat
                          BindingManager.AddBinding("returnDate.SelectedDate",
             }
             // event handler for findFlights button, uses injected
            // service and model 'trip' object to find flights
            private void SearchForFlights(object sender, EventArgs e
                          FlightSuggestions suggestions = bookingAgent.Suggest
                          if (suggestions.HasOutboundFlights)
                          {
                                      // redirect to SuggestedFlights page
                          }
             }
}
```

There are quite a few things that are happening in this relatively simple piece of code, so it's worth that we spend some time on each one:

1. When the page is initially loaded (IsPostback == false), the

InitializeModel() method is called which initializes the trip object by creating a new instance and setting its properties to desired values. Right before the page is rendered, the SaveModel() method will be invoked and whatever the value it returns will be stored within the HTTP Session. Finally, on each postback, the LoadModel() method will be called and the value returned by the previous call to SaveModel will be passed to it as an argument.

In this particular case the implementation is very simple because our whole model is just the trip object. As such, SaveModel() simply returns the trip object and LoadModel() casts the SavedModel() argument to Trip and assigns it to the trip field within the page. In the more complex scenarios, you will typically return a dictionary containing your model objects from the SaveModel() method, and read the values from that dictionary within the LoadModel().

2. InitializeDataBindings method defines the binding rules for all five input controls on our form. It does so by invoking AddBinding method on the BindingManager exposed by the page. AddBinding method is heavily overloaded and it allows you to specify a binding direction and a formatter to use in addition to the source and target binding expressions that are used above. We'll discuss these optional parameters shortly, but for now let's focus on the source and target expressions.

The Data Binding framework uses Spring.NET Expression Language to define binding expressions. In most cases, like in the example above, both source and target expression will evaluate to a property or a field within one of the controls or a data model. This is always the case when you are setting a bi-directional binding, as both binding expressions need to be "settable". What is important to remember about InitializeDataBindings method is that it is executed only once *per page type*. Basically, all of the binding expressions are parsed the first time the page is instantiated, and are the cached and used by all instances of that same page type that are created at a later time. This is done for performance reasons, as data binding expression parsing on every postback is unnecessary and would add a significant overhead to the overall page processing time.

3. If you look at the SearchForFlights event handler, you will notice that it has no dependencies on the view elements. It simply uses the injected bookingAgent service and a trip object that in order to obtain a list of suggested flights. Furthermore, if you make any modifications to the trip object within your event handler, bound controls will be updated accordingly just before the page is rendered.

This accomplishes one of the major goals we set out to achieve, allowing developers to remove view element references from the page event handlers and decouple controller-type methods from the view.

Now that you have a solid high-level picture of how Spring.NET data binding and model management are typically used in web applications, let's take a look at the details and see how data binding is actually implemented under the hood, what the extension points are, and what are some additional features that make data binding framework usable in real-world applications.

22.7.1. Data Binding Under the Hood

Spring.NET Data Binding framework revolves around two main interfaces: IBinding and IBindingContainer. The IBinding interface is definitely the more important one of the two, as it has to be implemented by all binding types. This interface defines several methods, with some of them being overloaded for convenience:

As their names imply, BindSourceToTarget method is used to extract

and copy bound values from the source object to the target object, while <code>BindTargetToSource</code> does the opposite. Both method names and parameter types are very generic for a good reason -- data binding framework can indeed be used to bind any two objects. Using it to bind web forms to model objects is just one of its possible uses, although a very common one and tightly integrated into the Spring.NET Web Framework.

The ValidationErrors parameter requires further explanation. While the data binding framework is not in any way coupled to the data validation framework, they are in some ways related. For example, while the data validation framework is best suited to validate the populated model according to the business rules, the data binding framework is in a better position to validate data types during the binding process. However, regardless of where specific validation is performed, all error messages should be presented to the user in a consistent manner. In order to accomplish this, Spring.NET Web Framework passes the same ValidationErrors instance to binding methods and to any validators that might be executed within your event handlers. This ensures that all error messages are stored together and are displayed consistently to the end user, using Spring.NET validation error controls.

The last method in the <code>IBinding</code> interface, <code>SetErrorMessage</code>, enables this by allowing you to specify the resource id of the error message to be displayed in the case of binding error, as well as the list of error providers that messages should be displayed in. We will see an example of the <code>SetErrorMessage</code> usage shortly.

The IBindingContainer interface extends the IBinding interface and adds the following members:

As you can see, this interface has a number of overloaded AddBinding methods. The first one, AddBinding(IBinding binding) is the most generic one, as it can be used to add any binding type to the container. The other four are convenience methods that provide a simple way to add the most commonly used binding type, SimpleExpressionBinding. The SimpleExpressionBinding is what we used in the example at the beginning of this section to bind our web form to a Trip instance. It uses Spring.NET Expression Language to extract and to set values within source and target objects. We discussed SourceExpression and targetExpression arguments earlier, so let's focus on the remaining ones.

22.7.1.1. Binding Direction

The direction argument determines whether the binding is bidirectional or unidirectional. By default, all data bindings are bidirectional unless the direction argument is set to either <code>BindingDirection.SourceToTarget</code> or <code>BindingDirection.TargetToSource</code>. If one of these two values is specified, binding will be evaluated only when the appropriate <code>BindDirection</code> method is invoked, and will be completely ignored in the other direction. This is very useful when you want to bind some information from the model into non-input controls, such as labels.

However, unidirectional data bindings are also useful when your form doesn't have a simple one-to-one mapping to presentation model. In our earlier trip form example, the presentation model was intentionally designed to allow for simple one-to-one mappings. For the sake of discussion, let's add the <code>Airport</code> class and modify our <code>TripPoint</code> class like this:

```
namespace SpringAir.Domain
{
    [Serializable]
    public class TripPoint
    {
        // fields
        private Airport airport;
        private DateTime date;
```

```
// constructors
    public TripPoint()
    {}
    public TripPoint(Airport airport, DateTime date)
    {
        this.airport = airport;
        this.date = date;
    }
    // properties
    public Airport Airport
    {
        get { return this.airport; }
        set { this.airport = value; }
    }
    public DateTime Date
        get { return this.date; }
        set { this.date = value; }
    }
}
[Serializable]
public class Airport
{
    // fields
    private string code;
    private string name;
   // properties
    public string Code
    {
        get { return this.code; }
        set { this.code = value; }
    }
    public string Name
        get { return this.name; }
        set { this.name = value; }
```

```
}
}
```

Instead of the string property AirportCode, our TripPoint class now exposes an Airport property of type Airport, which is defined above. Now we have a problem: what used to be a simple string to string binding, with the airport code selected in a dropdown being copied directly into the TripPoint.AirportCode property and vice versa, now becomes a not so simple string to Airport binding, so let's see how we can solve this mismatch problem.

First of all, binding from the model to the control is still very straight forward. We just need to set up one-way bindings from the model to controls:

All need to do is airport code value from the extract Trip.StartingFrom.Airport.Code of instead Trip.StartingFrom.AirportCode. Unfortunately, binding from the control to the model the same way won't work: we might be able to set Code property of the Airport object, but that will likely make the Airport. Name property invalid. What we really want do is find an instance of the Airport class based on the airport code and set TripPoint. Airport property to it. Fortunately, this is very simple to do with Spring.NET data binding, especially because we already have airportDao object defined in the Spring context, which GetAirport(string airportCode) finder method. All we need to do is set up data bindings from source to target that will invoke this finder method when evaluating the source expression. Our complete set of bindings for these two drop down lists will then look like this:

```
protected override void InitializeDataBindings()
```

```
{
    BindingManager.AddBinding("@(airportDao).GetAirport(
    BindingManager.AddBinding("leavingFromAirportCode.Set

    BindingManager.AddBinding("@(airportDao).GetAirport(
    BindingManager.AddBinding("goingToAirportCode.Select
    ...
}
```

That's it -- by using two unidirectional bindings with different expressions and by leveraging the fact that expressions can reference objects defined in the Spring context, we were able to solve this non-trivial data binding problem.

22.7.1.2. Formatters

The last argument to AddBinding method that we need to discuss is a formatter argument. This argument allows you to specify a formatter that should be used to parse string value from the typical input control before it is bound to the model, and to format strongly typed model value before it is bound to the control.

You will typically use one of the formatters provided in the Spring.Globalization.Formatters namespace, but if you have requirements that cannot be satisfied by one of the standard formatters it is easy enough to write your own -- all you need to do is implement a very simple IFormatter interface:

```
public interface IFormatter
{
    string Format(object value);
    object Parse(string value);
}
```

Standard formatters provided with Spring.NET are: CurrencyFormatter, DateTimeFormatter, FloatFormatter, IntegerFormatter, NumberFormatter and PercentFormatter, which should be sufficient for most usage scenarios.

22.7.1.3. Type Conversion

Because the data binding framework uses the same expression evaluation engine as the Spring.NET IoC container, it will use any registered type converters to perform data binding. Many type converters are included with Spring.NET (take a look at the classes in Spring.Objects.TypeConverters namespace) and automatically registered for you, but you can implement your own custom converters and register them using standard Spring.NET type converter registration mechanisms.

22.7.1.4. Data Binding Events

Spring.Web's base Page class adds two events to the standard .NET page lifecycle - DataBound and DataUnbound.

The **DataUnbound** event is fired after the data model has been updated using values from the controls. It is fired right after the **Load** event and only on postbacks, because it doesn't make sense to update the data model using the controls' initial values.

The **DataBound** is fired after controls have been updated using values from the data model. This happens right before the **PreRender** event.

The fact that data model is updated immediately after the Load event and that controls are updated right before the PreRender event means that your event handlers will be able to work with a correctly updated data model, as they execute after the Load event, and that any changes you make to the data model within event handlers will be reflected in the controls immediately afterwards, as they (the controls) are updated prior to the actual rendering.

22.7.1.5. Rendering Binding Errors

If there are errors in the databinding, for example, trying to bind a string 'hello' to an integer property on the model, you can specify how those fundamental binding errors should be rendered. An example of this shown below taken from the WebQuickStart 'RobustEmployeeInfo' example.

```
[Default.aspx.cs]
protected override void InitializeDataBindings()
{
   // collect txtId.Text binding errors in "id.errors" collect
```

```
BindingManager.AddBinding("txtId.Text", "Employee.Id").Set
[Default.aspx]
<asp:TextBox ID="txtId" runat="server" />
<!-- output validation errors from "id.errors" collection --
<spring:ValidationError Provider="id.errors" runat="server"</pre>
```

The SetErrorMessage specifies the message text or resource id of the error message to be displayed followed by a variable length list of strings that specify the collection of error providers message where the message should be displayed. In the above case the error provider will be rendered in Spring's ValidationError User Control. See

22.7.1.6. HttpRequestListBindingContainer

HttpRequestListBindingContainer extracts posted raw values from the request and populates the specified IList by creating objects of the type specified and populating each of these objects according to the requestBindings collection.

Please checkout the WebQuickStart of sample's demo HttpRequestListBindingContainer. Below

```
protected override void InitializeDataBindings()
  // HttpRequestListBindingContainer unbinds specified value
  HttpRequestListBindingContainer requestBindings =
      new HttpRequestListBindingContainer("sku, name, quantity
  requestBindings.AddBinding("sku", "Sku");
  requestBindings.AddBinding("name", "Name");
  requestBindings.AddBinding("quantity", "Quantity", quantit
  requestBindings.AddBinding("price", "Price", priceFormatte
  BindingManager.AddBinding(requestBindings);
}
```



Due to the fact, that browsers don't send the values of unchecked checkboxes, you can't use

HttpRequestListBindingContainer with <input type="checkbox" > html controls.

22.7.2. Using DataBindingPanel

To simplify use of Spring's Data Binding feature on web pages and controls, Spring.Web provides a special DataBindingPanel container control. A DataBindingPanel does not render any html code itself, but allows for specifying additional, data binding related attributes to its child controls:

```
<%@ Page Language="C#" CodeFile="Default.aspx.cs" Inherits="</pre>
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Cor</pre>
<html>
<body>
<spring:DataBindingPanel ID="ctlDataBindingPanel" runat="ser</pre>
   Employee ID:
          <asp:TextBox ID="txtId" runat="server" Bindi</pre>
          First Name:
          <asp:TextBox ID="txtFirstName" runat="server"
      </spring.DataBindingPanel>
</body>
</html>
```

Using DataBindingPanel the binding information can be specified directly on the control declaration. The following attributes are recognized by a DataBindingPanel:

- BindingTarget
 corresponds to the target expression used in IBindingContainer.AddBinding()
- BindingSource
 corresponds to the source expression used in

IBindingContainer.AddBinding(). For standard controls you don't need to specify the source expression. If you are binding to some custom control, of course you must specific this attribute.

- BindingDirection one of the values of the BindingDirection enumeration
- BindingFormatter

if you need a custom formatter, you can specific the object name of a formatter here. The formatter instance will be obtained by a call to IApplicationContext.GetObject() each time it is needed.

BindingType

In case you need a completely customized binding, specify its type here. Note that a custom binding type must implement the following constructor signature:

ctor(string source, string target, BindingDirection, IFormatter)



The Visual Studio Web Form Editor will of course complain about binding attributes because it doesn't know them. You can safely ignore those warnings.

22.7.3. Customizing Model Persistence

As was already mentioned in the introduction to this chapter, model management needs an application developer to override InitializeModel(), SaveModel() and LoadModel() for storing model information between requests in the user's session. On web farms this of course storing information in a user's session is not a good strategy. Thus it is possible to choose another persistence strategy by setting a Spring.Web.UI.Page's resp. Spring.Web.UI.UserControl's ModelPersistenceMedium property:

```
<object id="modelPersister" type="Sample.DatabaseModelPersis"</pre>
<object type="UserRegistration.aspx">
    property name="ModelPersistenceMedium" ref="modelPersis"
</object>
```

To implement any arbitrary persistence strategy, one simply needs to implement the IModelPersistenceMedium interface:

```
public interface IModelPersistenceMedium
{
    // Load the model for the specified control context.
    object LoadFromMedium( Control context );

    // Save the specified model object.
    void SaveToMedium( Control context, object modelToSave );
}
```

22.8. Localization and Message Sources

While recognizing that the .NET framework has excellent support for localization, the support within ASP.NET 1.x is somewhat incomplete.

Every .aspx page in an ASP.NET project has a resource file associated with it, but those resources are never used (by the current ASP.NET infrastructure). ASP.NET 2.0 will change that and allow application developers to use local resources for pages. In the meantime, the Spring.NET team built support for using local pages resources into Spring. Web thus allowing application developers to start using ASP.NET 2.0-like page resources immediately.

Spring. Web supports several different approaches to localization within a web application, which can be mixed and matched as appropriate. Both push and pull mechanisms are supported, as well as the fallback to globally defined resources when a local resource cannot be found. Spring. Web also provides support for user culture management and image localization, which are described in the later sections.



Introductory material covering ASP.NET Globalization and Localization can be found at the following URLs; Globalization Architecture for ASP.NET and Localization Practices for ASP.NET 2.0 by Michele Leroux Bustamante.

22.8.1. Automatic Localization Using Localizers ("Push" Localization)

The central idea behind 'push' localization is that an application developer should be able to specify localization resources in the resource file for the page and have those resources automatically applied to the user controls on the page by the framework. For example, an application developer could define a page such as UserRegistration.aspx...

```
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Cor</pre>
<%@ Page language="c#" Codebehind="UserRegistration.aspx.cs'</pre>
    AutoEventWireup="false" Inherits="ArtFair.Web.UI.Forms.L
<html>
    <body>
        <spring:Content id="mainContent" contentPlaceholderI</pre>
```

```
<div align="right">
              <asp:LinkButton ID="english" Runat="server"</pre>
              <asp:LinkButton ID="serbian" Runat="server"</pre>
          </div>
          <asp:Label id="emailLabel" Runat="se
                 <asp:TextBox id="email" Runat="serve
              <asp:Label id="passwordLabel" Runat=
                 <asp:TextBox id="password" Runat="se
              <tr>
                 <asp:Label id="passwordConfirmation|
                 <asp:TextBox id="passwordConfirmation"
              <asp:Label id="nameLabel" Runat="ser
                 <asp:TextBox id="name" Runat="server"
              . . .
              <asp:Button id="saveButton" Runat="s</pre>
                     <asp:Button id="cancelButton" Runat=</pre>
                 </spring:Content>
   </body>
</html>
```

A close inspection of the above .aspx code reveals that none of the Label or Button controls have had a value assigned to the Text property. The values of the Text property for these controls are stored in the local resource file (of the page) using the following convention to identify the resource (string).

```
<root>
  <data name="$this.emailLabel.Text">
    <value>Email:</value>
  </data>
  <data name="$this.passwordLabel.Text">
    <value>Password:</value>
  </data>
  <data name="$this.passwordConfirmationLabel.Text">
    <value>Confirm password:</value>
  </data>
  <data name="$this.nameLabel.Text">
    <value>Full name:</value>
  </data>
  <data name="$this.countryLabel.Text">
    <value>Country:</value>
  </data>
  <data name="$this.saveButton.Text">
    <value>$messageSource.save</value>
  </data>
  <data name="$this.cancelButton.Text">
    <value>$messageSource.cancel</value>
  </data>
</root>
```



VS2003

To view the .resx file for a page, you may need to enable "Project/Show All Files" in Visual Studio. When "Show All Files" is enabled, the .resx file appears like a "child" of the code-behind page.

When Visual Studio creates the .resx file, it will include a xds:schema element and several reshead elements. Your data elements will follow the reshead elements. When working with the .resx files, you may want to choose "Open With" from the context menu and select the "Source Code" text editor.



To create a resource file in VS2005, open your control or page in design mode and select "Tools/Generate local resource" from the menu

Finally a localizer must be configured for the page to enable automatic localization:

For more information on configuring localizers see <u>Section 22.8.3, "Working with Localizers"</u>

22.8.2. Global Message Sources

The last two resource definitions from the previous section require some additional explanation:

In some cases it makes sense to apply a resource that is defined *globally* as opposed to locally. In this example, it makes better sense to define values for the Save and Cancel buttons globally as they will probably be used throughout the application.

The above example demonstrates how one can achieve that by defining a *resource redirection* expression as the value of a local resource by prefixing a global resource name with the following string.

```
$messageSource.
```

Taking the case of the above example, this will tell the localizer to use the Save and Cancel portions of the resource key as lookup keys to retrieve the actual values from a global message source. The important thing to remember is that one need only define a resource redirect once, typically in the invariant resource file – any lookup for a resource redirect will simply fall back to the invariant culture, and result in a global message source lookup using the correct culture.

Global resources are (on a per-context basis) defined as a plain vanilla object definition using the reserved name of 'messageSource', which one can add to one's Spring.NET configuration file as shown below.



NET 2.0

To use resources from your App_GlobalResources folder, specify App_GlobalResources as assembly name (see the SpringAir example application for more):

<value>Resources.Strings,
App_GlobalResources

The global resources are cached within the Spring.NET IApplicationContext and are accessible through the Spring.NET IMessageSource interface.

The Spring.Web Page and UserControl classes have a reference to their owning IApplicationContext and it's associated IMessageSource. As such, they will automatically redirect resource lookups to a global message source if a local resource cannot be found.

Currently, the ResourceSetMessageSource is the only message source implementation that ships with Spring.NET.

22.8.3. Working with Localizers

In order to apply resources automatically, a localizer needs to be injected into all pages requiring this feature (typically accomplished using a base page definition that other pages will inherit from). The injected localizer will inspect the resource file when the page is first requested, cache the resources that start with the '\$this' marker string value, and apply the values to the controls that populate the page prior to the page being rendered.

A localizer is simply an object that implements the Spring.Globalization.ILocalizer interface. Spring.Globalization.AbstractLocalizer is provided as a convenient base class for localization: this class has one abstract method, LoadResources. This method must load and return a list of all the resources that must be automatically applied from the resource store.

Spring.NET ships with one concrete implementation of a localizer, Spring.Globalization.Localizers.ResourceSetLoca that retrieves a list of resources to apply from the local resource file. Future releases of Spring.NET may provide other localizers that read resources from an XML file or even a flat text file that contains resource name-value pairs which will allow application developers to store resources within the files in a web application instead of as embedded resources in an assembly. Of course, if an application developer would rather store such resources in a database, he or she can write their own ILocalizer implementation that will load a list of resources to apply from a database.

As mentioned previously, one would typically configure the localizer to be used within an abstract base definition for those pages that require localization as shown below.

Of course, nothing prevents an application developer from defining a different localizer for each page in the application; in any case, one can always override the localizer defined in a base (page) definition. Alternatively, if one does want any resources to be applied automatically one can completely omit the localizer definition.

One last thing to note is that Spring.NET **UserControl** instances will (by default) inherit the localizer and other localization settings from the page that they are contained within, but one can similarly also override that behavior using explicit dependency injection.

22.8.4. Applying Resources Manually ("Pull" Localization)

While automatic localization as described above works great for many form-like pages, it doesn't work nearly as well for the controls defined within any iterative controls because the IDs for such iterative controls are not fixed. It also doesn't work well in those cases where one needs to display the same resource multiple times within the same page. For example, think of the header columns for outgoing and return flights tables within the SpringAir application (see Chapter 37, SpringAir - Reference Application).

In these situations, one should use a pull-style mechanism for localization, which boils down to a simple **GetMessage** call as shown below.

```
<asp:Repeater id="outboundFlightList" Runat="server">
 <HeaderTemplate>
   <table border="0" width="90%" cellpadding="0" cellspacing
    <thead>
     <%= GetMessage("outboundFlights") %>
       <%= GetMessage("flightNumber") %>
       <%= GetMessage("departureDate") %>
       <%= GetMessage("departureAirport") %>
       <%= GetMessage("destinationAirport") %>
       <%= GetMessage("aircraft") %>
       <%= GetMessage("seatPlan") %>
```

The GetMessage method is available within both the Spring.Web.UI.Page and Spring.Web.UI.UserControl classes, and it will automatically fall back to a global message source lookup if a local resource is not found.

22.8.5. Localizing Images within a Web Application

Spring.Web provides an easy (and consistent) way to localize images within a web application. Unlike text resources, which can be stored within embedded resource files, XML files, or even a database, images in a typical web application are usually stored as files on the file system. Using a combination of directory naming conventions and a custom ASP.NET control, Spring.Web allows application developers to localize images within the page as easily as text resources.

The Spring.Web Page class exposes the ImagesRoot property, which is used to define the root directory where images are stored. The default value is 'Images', which means that the localizer expects to find an 'Images' directory within the application root, but one can set it to any value in the definition of the page.

In order to localize images, one needs to create a directory for each localized culture under the <code>ImagesRoot</code> directory as shown below.

```
/MyApp
/Images
/en
/en-US
/fr
/fr-CA
/sr-SP-Cyrl
/sr-SP-Latn
```

Once an appropriate folder hierarchy is in place all one need do is put the

localized images in the appropriate directories and make sure that different translations of the same image are named the same. In order to place a localized image on a page, one needs to use the Spring:LocalizedImage> as shown below.

This control will find the most specific directory that contains an image with the specified name using standard localization fallback rules and the user's culture. For example, if the user's culture is 'en-US', the localizer will look for the spring-air-logo.jpg file in Images/en-US, then in Images/en and finally, if the image file has still not been found, in the root Images directory (which for all practical purposes serves as an invariant culture folder).

22.8.6. User Culture Management

In addition to global and local resource management, Spring.Web also adds support for user culture management by exposing the current CultureInfo through the UserCulture property on the Page and UserControl classes.

The UserCulture property will simply delegate culture resolution to an implementation of

Spring.Globalization.ICultureResolver interface. One can specify exactly which culture resolver to use by configuring the CultureResolver property of the Page class in the relevant object definition as shown below.

```
<object name="BasePage" abstract="true">
  cobject name="CultureResolver">
```

Several useful implementations of <code>ICultureResolver</code> ship as part of Spring.Web, so it is unlikely that application developers will have to implement their own culture resolver. However, if one does have such a requirement, the resulting implementation should be fairly straightforward as there are only two methods that one need implement. The following sections discuss each available implementation of the <code>ICultureResolver</code> interface.

22.8.6.1. DefaultWebCultureResolver

This is default culture resolver implementation. It will be used if one does not specify a culture resolver for a page, or if one explicitly injects a <code>DefaultWebCultureResolver</code> into a page definition explicitly. The latter case (explicit injection) is sometimes useful because it allows one to specify a culture that should always be used by providing a value to the <code>DefaultCulture</code> property on the resolver.

The <code>DefaultWebCultureResolver</code> will first look at the <code>DefaultCulture</code> property and return its value if said property value is not null. If it is null, the <code>DefaultWebCultureResolver</code> will fall back to request header inspection, and finally, if no <code>'Accept-Lang'</code> request headers are present it will return the UI culture of the currently executing thread.

22.8.6.2. RequestCultureResolver

This resolver works in a similar way to the <code>DefaultWebCultureResolver</code> with the exception that it always checks request headers <code>first</code>, and only then falls back to the value of the <code>DefaultCulture</code> property or the culture code of the current thread.

22.8.6.3. SessionCultureResolver

This resolver will look for culture information in the user's session and return it if it finds one. If not, it will fall back to the behavior of the DefaultWebCultureResolver.

22.8.6.4. CookieCultureResolver

This resolver will look for culture information in a cookie, and return it if it finds one. If not, it will fall back to the behavior of the DefaultWebCultureResolver.

Warning

CookieCultureResolver will not work if your application uses localhost as the server URL, which is a typical setting in a development environment.

In order to work around this limitation you should use SessionCultureResolver during development and switch to CookieCultureResolver before you deploy the application in a production. This is easily accomplished in Spring.Web (simply change the config file) but is something that you should be aware of.

22.8.7. Changing Cultures

In order to be able to change the culture application developers will need to use one of the culture resolvers that support culture changes, such as SessionCultureResolver or CookieCultureResolver. One could also write a custom ICultureResolver that will persist culture information in a database, as part of a user's profile.

Once that requirement is satisfied, all that one need do is to set the UserCulture property to a new CultureInfo object before the page is rendered. In the following .aspx example, there are two link buttons that can be used to change the user's culture. In the code-behind, this is all one need do to set the new culture. A code snippet for the code-behind file (UserRegistration.aspx.cs) is shown below.

```
protected override void OnInit(EventArgs e)
{
    InitializeComponent();

    this.english.Command += new CommandEventHandler(this.Set this.serbian.Command += new CommandEventHandler(this.Set
```

```
base.OnInit(e);
}

private void SetLanguage(object sender, CommandEventArgs e)
{
    this.UserCulture = new CultureInfo((string) e.CommandArg)}
```

22.9. Result Mapping

One of the problems evident in many ASP.NET applications is that there is no built-in way to externalize the flow of an application. The most common way of defining application flow is by hardcoding calls to the Response.Redirect and Server.Transfer methods within event handlers.

This approach is problematic because any changes to the flow of an application necessitates code changes (with the attendant recompilation, testing, redeployment, etc). A much better way, and one that has been proven to work successfully in many MVC (Model-View-Controller) web frameworks is to provide the means to externalize the mapping of action results to target pages.

Spring. Web adds this functionality to ASP.NET by allowing one to define result mappings within the definition of a page, and to then simply use logical result names within event handlers to control application flow.

In Spring.Web, a logical result is encapsulated and defined by the Result class; because of this one can configure results just like any other object:

```
<objects xmlns="http://www.springframework.net">
    <object id="homePageResult" type="Spring.Web.Support.Res</pre>
        cproperty name="TargetPage" value="~/Default.aspx"/>
        property name="Mode" value="Transfer"/>
        property name="Parameters">
            <dictionary>
                <entry key="literal" value="My Text"/>
                <entry key="name" value="%{UserInfo.FullName</pre>
                <entry key="host" value="%{Request.UserHostN</pre>
            </dictionary>
        </property>
    </object>
    <object id="loginPageResult" type="Spring.Web.Support.Re</pre>
        property name="TargetPage" value="Login.aspx"/>
        property name="Mode" value="Redirect"/>
    </object>
    <object type="UserRegistration.aspx" parent="basePage">
        cproperty name="UserManager" ref="userManager"/>
```

```
property name="Results">
             <dictionary>
                 <entry key="userSaved" value-ref="homePageRe</pre>
                 <entry key="cancel" value-ref="loginPageResu</pre>
             </dictionary>
        </property>
    </object>
</objects>
```

The only property that you *must* supply a value for each and every result is the TargetPage property. The value of the Mode property can be either Transfer, TransferNoPreserve, Redirect, and defaults to Transfer if none is specified. TransferNoPreserve issues a server-side transfer with 'preserveForm=false', so that QueryString and Form data are not preserved.

If one's target page requires parameters, one can define them using the Parameters dictionary property. One simply specifies either literal values or object navigation expressions for such parameter values; if one specifies an expression, this expression will be evaluated in the context of the page in which the result is being referenced... in the specific case of the above example, this means that any page that uses the homePageResult needs to expose a **UserInfo** property on the page class itself.



In Spring 1.1.0 and before the prefix used to indicate an object navigation expression in the Parameters dictionary property was the dollar sign, i.e. \${UserInfo.FullName}. This conflicted with the prefix used to perform property replacement, the dollar sign, as described in the section PropertyPlaceholderConfigurer. As a workaround you can change the prefix and suffix used in PropertyPlaceholderConfigurer to be different, for example prefix = \$\${ and suffix = }. In Spring 1.1.1 a new prefix character, the percent sign (i.e.%{UserInfo.FullName}.) can be used in the Parameters dictionary to avoid this conflict so you can keep the familiar NAnt style PropertyPlaceholderConfigurer defaults.

Parameters will be handled differently depending on the result mode. For

redirect results, every parameter will be converted to a string, then URL encoded, and finally appended to a redirect query string. On the other hand, parameters for transfer results will be added to the <code>HttpContext.Items</code> collection before the request is transferred to the target page. This means that transfers are more flexible because any object can be passed as a parameter between pages. They are also more efficient because they don't require a round-trip to the client and back to the server, so transfer mode is recommended as the preferred result mode (it is also the current default).



Tip

If you should need to customize how to a redirect request is generate, for example to encrypt the request parameters, subclass the Request object and override one or more protected methods, for example string BuildUrl(string resolvedPath, IDictionary resolvedParameters). See the API documentation for additional information.

The above example shows independent result object definitions, which are useful for global results such as a home- and login- page. Result definitions that are only going to be used by one page should be simply embedded within the definition of a page, either as inner object definitions or using a special shortcut notation for defining a result definition:

The short notation for the result must adhere to the following format...

```
[<mode>:]<targetPage>[?param1,param2,...,paramN]
```

There are three possible values for the **mode** value referred to in the above notation snippet; they are:

- redirect
- transfer
- TransferNoPreserve

They correspond to the values of the ResultMode enumeration. One thing to notice is that a comma is used instead of an ampersand to separate parameters; this is done so as to avoid the need for laborious ampersand escaping within an XML object definition. The use of the ampersand character is still supported if required, but one will then have to specify it using the well known & entity reference.

Once one has defined one's results, it is very simple to use them within the event handlers of one's pages (UserRegistration.apsx.cs)...

```
private void SaveUser(object sender, EventArgs e)
{
    UserManager.SaveUser(UserInfo);
    SetResult("userSaved");
}
public void Cancel(object sender, EventArgs e)
{
    SetResult("cancel");
}
protected override void OnInit(EventArgs e)
{
    InitializeComponent();
    this.saveButton.Click += new EventHandler(this.SaveUser)
    this.cancelButton.Click += new EventHandler(this.Cancel)
    base.OnInit(e);
}
```

One could of course further refactor the above example and use defined constants. This would be a good thing to do in the case of a logical result name such as "home" that is likely to be referenced by many pages.

22.9.1. Registering user defined transfer modes

You may also register a custom interpreter that can parse the short-hand string representation that creates a Result object.. The string representation can be broken down into two parts shown below

```
<resultmode>:<textual result representation>
```

The interface IResultFactory is responsible for creating an IResult object from these two pieces, as shown below

```
public interface IResultFactory
{
    IResult CreateResult( string resultMode, string resultTe
}
```

A ResultFactoryRegistry is used to associate a given resultmode string with an IResultFactory implementation. Here is an example

```
class MySpecialResultLogic : IResult
{
    ...
}

class MySpecialResultLogicFactory : IResultFactory
{
    IResult Create( string mode, string expression ) {
        /* ... convert 'expression' into MySpecialResultLogic *
    }
}

// register with global factory
ResultFactoryRegistry.RegisterResultFactory( "mySpecialMode"
```

You would then use the custom 'continue' mode in your page as shown below

The result redirection is done as before, by calling myPage.SetResult("continue");

22.10. Client-Side Scripting

ASP.NET has decent support for client-side scripting through the use of the Page.RegisterClientScriptBlock and Page.RegisterStartupScript methods.

However, neither of these two methods allows you to output a registered script markup within a <head> section of a page, which is (in many cases) exactly what you would like to do.

22.10.1. Registering Scripts within the head HTML section

Spring.Web.UI.Page class: RegisterHeadScriptBlock and RegisterHeadScriptFile, each with a few overrides. You can call these methods from your custom pages and controls in order to register script blocks and script files that must be included in the <head> section of the final HTML page.

The only additional thing that is required to make this work is that you use the <spring:Head> server-side control to define your <head> section
instead of using the standard HTML <head> element. This is shown below.

```
<%@ Page language="c#" Codebehind="StandardTemplate.aspx.cs"</pre>
                                                                           AutoEventWireup="false" Inherits="SpringAir.
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Cor</pre>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN</pre>
<html>
         <spring:Head runat="server" id="Head1">
                   <title>
                             <spring:ContentPlaceHolder id="title" runat="server">
                                      <%= GetMessage("default.title") %>
                            </spring:ContentPlaceHolder>
                   </title>
                   <LINK href="<%= CssRoot %>/default.css" type="text/css"
                   <spring:ContentPlaceHolder id="head" runat="server"></spring:ContentPlaceHolder 
         </spring:Head>
         <body>
         </body>
</html>
```

The example above shows you how you would typically set-up a <head> section within a master page template in order to be able to change the title value and to add additional elements to the <head> section from the child pages using <spring:ContentPlaceholder> controls. However, only the <spring:Head> declaration is required in order for Spring.NET Register* scripts to work properly.

22.10.2. Adding CSS Definitions to the head Section

In a similar fashion, you can add references to CSS files, or even specific styles, directly to the <head> HTML section using Page.RegisterStyle and Page.RegisterStyleFile methods. The latter one simply allows you to include a reference to an external CSS file, while the former one allows you to define embedded style definitions by specifying the style name and definition as the parameters. The final list of style definitions registered this way will be rendered within the single embedded <code>Style</code> section of the final HTML document.

22.10.3. Well-Known Directories

In order to make the manual inclusion of client-side scripts, CSS files and images easier, the Spring.Web Page class exposes several properties that help you reference such artifacts using absolute paths. This affords web application developers a great deal of convenience functionality straight out of the box if they stick to common conventions such as a web application (directory) structure..

These properties are ScriptsRoot, CssRoot and ImagesRoot. They have default values of Scripts, CSS and Images, which will work just fine if you create and use these directories in your web application root. However, if you prefer to place them somewhere else, you can always override default values by injecting new values into your page definitions (you will typically inject these values only in the base page definition, as they are normally shared by all the pages in the application). An example of such configuration is shown below:

22.11. Spring User Controls

Spring provides several custom user controls that are located in the Spring.Web.UI.Controls namespace. This section primarily lists the controls and points to other documentation to provide additional information. There are a few other controls not documented here, please check the SDK docs for their descriptions.

22.11.1. Validation Controls

The location in the web page where validation errors are to be rendered can be specifies by using the ValidationSummary and ValidationError controls. There are two controls since they have different defaults for how errors are rendered. ValidationSummary is used to display potentially multiple errors identified by the validation framework. ValidationError is used to display field-level validation errors. Please refer to the section ASP.NET usage tips in the chapter on the Validation Framework more information.

22.11.2. Databinding Controls

Some standard controls are not easy to use with Spring's databinding support. Examples are check boxes and ratio button groups. In this case you should use the CheckBoxList and RadioButtonGroup controls. Databinding itself can be done using the DataBindingPane instead of the using the BindingManager API within the code behind page.

22.11.3. Calendar Control

A pop-up DHTML calendar control is provided. It is a slightly modified version of the Dynarch.com DHTML Calendar control written by Mihai Bazon.

22.11.4. Panel Control

You can suppress dependency injection for controls inside your ASP.NET by using the Panel control. See the section <u>Customizing control dependency injection</u> for more information.

Chapter 23. ASP.NET AJAX

23.1. Introduction

Spring's ASP.NET AJAX integration allows for a plain .NET object (PONO), that is one that doesn't have any attributes or special base classes, to be exported as a web service, configured via dependency injection, 'decorated' by applying AOP, and then exposed to client side JavaScript.

23.2. Web Services

Spring.NET, and particularly Spring.Web, improved <u>support for web services</u> in .NET with the WebServiceExporter. Exporting of an ordinary plain .NET object as a web service is achieved by registering a custom implementation of the WebServiceHandlerFactory class as the HTTP handler for *.asmx requests.

Microsoft ASP.NET AJAX introduced a new HTTP handler System. Web. Script. Services. ScriptHandlerFactory to allow a Web Service to be invoked from the browser by using JavaScript.

Spring's integration allows for both Spring.Web and ASP.NET AJAX functionality to be used together by creating a new HTTP handler.

23.2.1. Exposing Web Services

The WebServiceExporter combined with the new HTTP handler exposes PONOs as Web Services in your ASP.NET AJAX application.

In order for a Web service to be accessed from script, the WebServiceExporter should decorate the Web Service class with the ScriptServiceAttribute. The code below is taken from the sample application Spring.Web.Extensions.Sample, aka the 'AJAX' shortcut in the installation.:

All that one needs to do in order to use the WebServiceExporter is:

1. Configure the Web.config file of your ASP.NET AJAX application as a Spring. Web application.

```
<sectionGroup name="spring">
  <section name="context" type="Spring.Context.Support.WebCo"
</sectionGroup>
```

```
<spring>
  <context>
        <resource uri="~/Spring.config"/>
        </context>
  </spring>
```

2. Register the HTTP handler and the Spring HttpModule under the **System.web** section.

```
<httpHandlers>
    <remove verb="*" path="*.asmx"/>
    <add verb="*" path="*.asmx" validate="false" type="Spring.
    <add verb="*" path="*_AppService.axd" validate="false" type="Syste"
    <add verb="GET, HEAD" path="ScriptResource.axd" type="Syste"
    </httpHandlers>
    <add name="ScriptModule" type="System.Web.Handlers.ScriptMeadd name="SpringModule" type="Spring.Context.Support.WebSet)
    </httpModules>
```

3. Register the HTTP handler and the Spring HttpModule under system.webServer section.

```
<modules>
  <add name="ScriptModule" preCondition="integratedMode" type="Spring.Context.Support.WebS")</pre>
```

You can find a full Web.config file in the example that comes with this integration.

23.2.2. Calling Web Services by using JavaScript

A proxy class is generated for each Web Service. Calls to Web Services methods are made by using this proxy class. When using the WebServiceExporter, the name of the proxy class is equal to the WebServiceExporter's id.

```
// This function calls the Contact Web service method
// passing simple type parameters and the callback function
function GetEmails(prefix, count)
{
    ContactWebService.GetEmails(prefix, count, GetEmailsOnSu)
}
```

Spring Framework



Part IV. Services

This part of the reference documentation covers the Spring Framework's integration with .NET distributed technologies such as .NET Remoting, Enterprise Services, Web Services. Integration with WCF Services is forthcoming. Please refer to the introduction chapter for more details.

- Chapter 24, Introduction to Spring Services
- Chapter 25, .NET Remoting
- Chapter 26, .NET Enterprise Services
- Chapter 27, Web Services
- Chapter 28, Windows Communication Foundation (WCF)

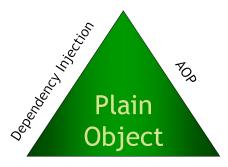
Chapter 24. Introduction to Spring Services

24.1. Introduction

The goal of Spring's integration with distributed technologies is to adapt plain .NET objects so they can be used with a specific distributed technology. This integration is designed to be as non-intrusive as possible. If you need to expose an object to a remote process then you can define an exporter for that object. Similarly, on the client side you define an corresponding endpoint accessor. Of course, the object's methods still need to be suitable for remoting, i.e. coarse grained, to avoid making unnecessary and expensive remote calls.

Since these exporters and client side endpoint accessors are defined using meta data for Spring IoC container, you can easily use dependency injection on them to set initial state and to 'wire up' the presentation tier, such as web forms, to the service layer. In addition, you may apply AOP aspects to the exported classes and/or service endpoints to apply behavior such as logging, security, or other custom behavior that may not be provided by the target distributed technology. The Spring specific terminology for this approach to object distribution is known as Portable Service Abstractions (PSA). As a result of this approach, you can decide much later in the development process the technical details of how you will distribute your objects as compared to traditional code centric approaches. Changing of the implementation is done though configuration of the IoC container and not by recompilation. Of course, you may choose to not use the IoC container to manage these objects and use the exporter and service endpoints programatically.

The diagram shown below is a useful way to demonstrate the key abstractions in the Spring tool chest and their interrelationships. The four key concepts are; plain .NET objects, Dependency Injection, AOP, and Portable Service Abstractions. At the heart sits the plain .NET object that can be instantiated and configured using dependency injection. Then, optionally, the plain object can be adapted to a specific distributed technology. Lastly, additional behavior can be applied to objects. This behavior is typically that which can not be easily address by traditional OO approaches such as inheritance. In the case of service layer, common requirements such as 'the service layer must be transactional' are implemented in a manner that naturally expresses that intention in a single place, as compared to scattered code across the service layer.



Portable Service Abstractions

Spring implements this exporter functionality by creating a proxy at runtime that meets the implementation requirements of a specific distributed technology. In the case of .NET Remoting the proxy will inherit from MarshalByRef, for EnterpriseServices it will inherit from ServicedComponent and for aspx web services, WebMethod attributes will be added to methods. Client side functionality is often implemented by a thin layer over the client access mechanism of the underlying distributed technology, though in some cases such as client side access to web services, you have the option to create a proxy on the fly from the .wsdl definition, much like you would have done using the command line tools.

The common implementation theme for you as a provider of these service objects is to implement an interface. This is generally considered a best practice in its own right, you will see most pure WCF examples following this practice, and also lends itself to a straightforward approach to unit testing business functionality as stub or mock implementations may be defined for testing purposes.

The assembly Spring.Services.dll contains support for <u>NET Remoting</u>, <u>Enterprise Services</u> and <u>ASMX Web Services</u>. Support for WCF services is planned for Spring 1.2 and is currently in the CVS repository if you care to take an early look.

Chapter 25. .NET Remoting

25.1. Introduction

Spring's .NET Remoting support allows you to export a 'plain .NET object' as a .NET Remoted object. By "plain .NET object" we mean classes that do not inherit from a specific infrastructure base class such as MarshalByRefObject. On the server side, Spring's .NET Remoting exporters will automatically create a proxy that implements MarshalByRefObject. You register SAO types as either SingleCall or Singleton and also configure on a per-object basis lifetime and leasing parameters. On the client side you can obtain CAO references to server proxy objects in a manner that promotes interface based design best practices when developing .NET remoting applications. The current implementation requires that your plain .NET objects implements a business service interface. Additionally you can add AOP advice to both SAO and CAO objects.

You can leverage the IoC container to configure the exporter and service endpoints. A remoting specific xml-schema is provided to simplify the remoting configuration but you can still use the standard reflection-like property based configuration schema. You may also opt to not use the IoC container to configure the objects and use Spring's .NET Remoting classes Programatically, as you would with any third party library.

A sample application, often referred to in this documentation, is in the distribution under the directory "examples\Spring\Spring.Calculator" and may also be found via the start menu by selecting the 'Calculator' item.

25.2. Publishing SAOs on the Server

Exposing a Singleton SAO service can be done in two ways. The first is through programmatic or administrative type registration that makes calls to RemotingConfiguration.RegisterWellKnownServiceT This method has the limitation that you must use a default constructor and you can not easily configure the singleton state at runtime since it is created on demand. The second way is to publish an object instance using RemotingServices.Marshal. This method overcomes the limitations of the first method. Example server side code for publishing an SAO singleton object with a predefined state is shown below

```
AdvancedMBRCalculator calc = new AdvancedMBRCalculator(217);
RemotingServices.Marshal(calc, "MyRemotedCalculator");
```

The class AdvancedMBRCalculator used above inherits from MarshalByRefObject.

If your design calls for configuring a singleton SAO, or using a non-default constructor, you can use the Spring IoC container to create the SAO instance, configure it, and register it with the .NET remoting infrastructure. The SaoExporter class performs this task and most importantly, will automatically create a proxy class that inherits from MarshalbyRefObject if your business object does not already do so. The following XML taken from the Remoting QuickStart demonstrates its usage to an SAO Singleton object

25.2.1. SAO Singleton

This XML fragment shows how an existing object "singletonCalculator" defined

the exposed under in Spring context the url-path is name "RemotedSaoSingletonCalculator". (The fully qualified is tcp://localhost:8005/RemotedSaoSingleCallCalculator using the standard .NET channel configuration shown further below.) AdvancedCalculator class implements the business interface IAdvancedCalculator. The current proxy implementation requires that your business objects implement an interface. The interfaces' methods will be the ones exposed in the generated .NET remoting proxy. The initial memory of the calculator is set to 217 via the constructor. The class AdvancedCalculator does not inherit from MarshalByRefObject. Also note that the exporter sets the lifetime of the SAO Singleton to infinite so that the singleton will not be garbage collected after 5 minutes (the .NET default lease time). If you would like to vary the they are InitialLeaseTime, RenewOnCallTime, lifetime properties, SponsorshipTimeout.

A custom schema is provided to make the object declaration even easier and with intellisense support for the attributes. This is shown below

Refer to the end of this chapter for more information on Spring's .NET custom schema.

25.2.2. SAO SingleCall

The following XML fragment shows how to expose the calculator service in SAO 'SingleCall' mode.

Note that we change the singleton attribute of the plain .NET object as configured by Spring in the <object> definition and not an attribute on the SaoExporter. The object referred to in the TargetName parameter can be an AOP proxy to a business object. For example, if we were to apply some simple logging advice to the singleton calculator, the following standard AOP configuration is used to create the target for the SaoExporter



Note 🔪

As generally required with a .NET Remoting application, the arguments to your service methods should be Serializable.

25.2.1. Console Application Configuration

When using SaoExporter you can still use the standard remoting administration section in the application configuration file to register the channel. ChannelServices as shown below

```
<system.runtime.remoting>
```

A console application that will host this Remoted object needs to initialize the .NET Remoting infrastructure with a call to RemotingConfiguration (since we are using the .config file for channel registration) and then start the Spring application context. This is shown below

```
RemotingConfiguration.Configure("RemoteApp.exe.config");
IApplicationContext ctx = ContextRegistry.GetContext();
Console.Out.WriteLine("Server listening...");
Console.ReadLine();
```

You can also put in the configuration file an instance of the object Spring.Remoting.RemotingConfigurer make RemotingConfiguration call show above on your behalf during initialization of The RemotingConfigurer IoC container. implements the IObjectFactoryPostProcessor interface, which gets called after all object definitions have been loaded but before they have been instantiated, (SeeSection 5.9.2, "Customizing configuration metadata with ObjectFactoryPostProcessors" for more information). The RemotingConfigurer has two properties you can configure. Filename, that specifies the filename to load the .NET remoting configuration from (if null the default file used) name is EnsureSecurity which makes sure the channel in encrypted (available only on .NET 2.0). As a convenience, the custom Spring remoting schema can be used to define an instance of this class as shown below, taken from the Remoting QuickStart

```
<objects xmlns="http://www.springframework.net"
    xmlns:r="http://www.springframework.net/remoting">
```

```
<r:configurer filename="Spring.Calculator.RemoteApp.exe.co"
</objects>
```

The ReadLine prevents the console application from exiting. You can refer to the code in RemoteApp in the Remoting QuickStart to see this code in action.

25.2.3. IIS Application Configuration

If you are deploying a .NET remoting application inside IIS there is a <u>sample</u> <u>project</u> that demonstrates the necessary configuration using Spring.Web.

Spring.Web ensures the application context is initialized, but if you don't use Spring.Web the idea is to start the initialization of the Spring IoC container inside the application start method defined in Global.asax, as shown below

In this example, the Spring configuration file is named Spring.Config. Inside Web.config you add a standard <system.runtime.remoting> section. Note that you do not need to specify the port number of your channels as they will use the port number of your web site. Ambiguous results have been reported if you do specify the port number. Also, in order for IIS to recognize the remoting request, you should add the suffix '.rem' or '.soap' to the target name of your exported remote object so that the correct IIS handler can be invoked.

25.3. Accessing a SAO on the Client

Administrative type registration on the client side lets you easily obtain a reference to a SAO object. When a type is registered on the client, using the new operator or using the reflection API will return a proxy to the remote object instead of a local reference. Administrative type registration on the client for a SAO object is performed using the wellknown element in the client configuration section. However, this approach requires that you expose the implementation of the class on the client side. Practically speaking this would mean linking in the server assembly to the client application, a generally recognized bad practice. This dependency can be removed by developing remote services based on a business interface. Aside from remoting considerations, the separation of interface and implementation is considered a good practice when designing OO systems. In the context of remoting, this means that the client can obtain a proxy to a specific implementation with *only* a reference to the interface assembly. To achieve the decoupling of client and server, a separate assembly containing the interface definitions is created and shared between the client and server applications.

There is a simple means for following this design when the remote object is a SAO object. A call to Activator.GetObject will instantiate a SAO proxy on the client. For CAO objects another mechanism is used and is discussed later. The code to obtain the SAO proxy is shown below

```
ICalculator calc = (ICalculator)Activator.GetObject (
   typeof (ICalculator),
   "tcp://localhost:8005/MyRemotedCalculator");
```

To obtain a reference to a SAO proxy within the IoC container, you can use the object factory SaoFactoryObject in the Spring configuration file. The following XML taken from the Remoting QuickStart demonstrates its usage.

```
<object id="calculatorService" type="Spring.Remoting.SaoFact
  <pre>cproperty name="ServiceInterface" value="Spring.Calculator
  cproperty name="ServiceUrl" value="tcp://localhost:8005/Recomplete">
```

The ServiceInterface property specifies the type of proxy to create while the

ServiceUrl property creates a proxy bound to the specified server and published object name.

Other objects in the IoC container that depend on an implementation of the interface <code>ICalculator</code> can now refer to the object "calculatorService", thereby using a remote implementation of this interface. The exposure of dependencies among objects within the IoC container lets you easily switch the implementation of <code>ICalculator</code>. By using the IoC container changing the application to use a local instead of remote implementation is a configuration file change, not a code change. By promoting interface based programing, the ability to switch implementation makes it easier to unit test the client application, since unit testing can be done with a mock implementation of the interface. Similarly, development of the client can proceed independent of the server implementation. This increases productivity when there are separate client and server development teams. The two teams agree on interfaces before starting development. The client team can quickly create a simple, but functional implementation and then integrate with the server implementation when it is ready.

25.4. CAO best practices

Creating a client activated object (CAO) is typically done by administrative type registration, either Programatically or via the standard .NET remoting configuration section. The registration process allows you to use the 'new' operator to create the remote object and requires that the implementation of the object be distributed to the client. As mentioned before, this is not a desirable approach to developing distributed systems. The best practice approach that avoids this problem is to create an SAO based factory class on the server that will return CAO references to the client. In a manner similar to how Spring's generic object factory can be used as a replacement creating a factory per class, we can create a generic SAO object factory to return CAO references to objects defined in Spring's application context. This functionality is encapsulated in Spring's CaoExporter class. On the client side a reference is obtained using CaoFactoryObject. The client side factory object supports creation of the CAO object using constructor arguments. In addition to reducing the clutter and tedium around creating factory classes specific to each object type you wish to expose in this manner, this approach has the additional benefit of not requiring any type registration on the client or server side. This is because the act of returning an instance of a class that inherits from MarshalByRefObject across a remoting boundary automatically returns a CAO object reference. For more information on this best-practice, refer to the last section, Section 25.8, "Additional Resources", for some links to additional resources.

25.5. Registering a CAO object on the Server

To expose an object as a CAO on the server you should declare an object in the standard Spring configuration that is a 'prototype', that is the singleton property is set to false. This results in a new object being created each time it is retrieved from Spring's IoC container. An implementation of ICaoRemoteFactory is what is exported via a call to RemotingServices.Marshal. This implementation uses Spring's IoC container to create objects and then dynamically create a .NET remoting proxy for the retrieved object. Note that the default lifetime of the remote object is set to infinite (null is returned from the implementation of InitializeLifetimeService()).

This is best shown using an example from the Remoting Quickstart application. Here is the definition of a simple calculator object,

To export this as a CAO object we can declare the CaoExporter object directly in the server's XML configuration file, as shown below

```
<object id="caoCalculator" type="Spring.Remoting.CaoExporter
  <pre>cproperty name="TargetName" value="prototypeCalculator" />
  cproperty name="Infinite" value="false" />
  cproperty name="InitialLeaseTime" value="2m" />
  cproperty name="RenewOnCallTime" value="1m" />
  </object>
```

Note the property 'TargetName' is set to the name, not the reference, of the non-singleton declaration of the 'AdvancedCalculator' class.

Alternatively, you can use the remoting schema and declare the CAO object as shown below

```
<r:caoExporter targetName="prototypeCalculator" infinite="fa
     <r:lifeTime initialLeaseTime="2m" renewOnCallTime="1m" />
</r:caoExporter>
```

25.5.1. Applying AOP advice to exported CAO objects

Applying AOP advice to exported CAO objects is done by referencing the adviced object name to the CAO exporter. Again, taking an example from the Remoting QuickStart, a calculator with logging around advice is defined as shown below.

If this declaration is unfamiliar to you, please refer to <u>Chapter 13</u>. <u>Aspect Oriented Programming with Spring.NET</u> for more information. The CAO exporter then references with the name 'prototypeCalculatorWeaved' as shown below.

```
<r:caoExporter targetName="prototypeCalculatorWeaved" infini
  <r:lifeTime initialLeaseTime="2m" renewOnCallTime="1m" />
</r:caoExporter>
```

25.6. Accessing a CAO on the Client

On the client side a CAO reference is obtained by using the CaoFactoryObject as shown below

This definition corresponds to the exported calculator from the previous section. The property 'RemoteTargetName' identifies the object on the server side. Using this approach the client can obtain an reference though standard DI techniques to a remote object that implements the <code>IAdvancedCalculator</code> interface. (As always, that doesn't mean the client should treat the object as if it was an inprocess object).

Alternatively, you can use the Remoting schema to shorten this definition and provide intellisense code completion

25.6.1. Applying AOP advice to client side CAO objects.

Applying AOP advice to a client side CAO object is done just like any other object. Simply use the id of the object created by the CaoFactoryObject as the AOP target, i.e. 'calculatorService' in the previous example.

25.7. XML Schema for configuration

Please install the XSD schemas into VS.NET as described in <u>Chapter 32</u>, <u>Visual Studio.NET Integration</u>. XML intellisense for the attributes of the saoExporter, caoExporter and caoFactory should be self explanatory as they mimic the standard property names used to configure .NET remote objects.

25.8. Additional Resources

Two articles that describe the process of creating a standard SAO factory for returning CAO objects are Implementing Broker with .NET Remoting Using Client-Activated Objects on MSDN and SAO class factories on Glacial Components website.

Chapter 26. .NET Enterprise Services

26.1. Introduction

Spring's .NET Enterprise Services support allows you to export a 'plain .NET object' as a .NET Remoted object. By "plain .NET object" we mean classes that do not inherit from a specific infrastructure base class such as ServicedComponent..

You can leverage the IoC container to configure the exporter and service endpoints. You may also opt to not use the IoC container to configure the objects and use Spring's .NET Enterprise Services classes Programatically, as you would with any third party library.

26.2. Serviced Components

Services components in .NET are able to use COM+ services such as declarative and distributed transactions, role based security, object pooling messaging. To access these services your class needs to derive from the class <code>System.EnterpriseServices.ServicedComponent</code>, adorn your class and assemblies with relevant attributes, and configure your application by registering your serviced components with the COM+ catalog. The overall landscape of accessing and using COM+ services within .NET goes by the name .NET Enterprise Services.

Many of these services can be provided without the need to derive from a ServicedComponent though the use of Spring's Aspect-Oriented Programming functionality. Nevertheless, you may be interested in exporting your class as a serviced component and having client access that component in a location transparent manner. By using Spring's

ServicedComponentExporter,

EnterpriseServicesExporter

and

ServicedComponentFactory you can easily create and consume serviced components without having your class inherit from ServicedComponent and automate the manual deployment process that involves strongly signing your assembly and using the regsvcs utility.

Note that the following sections do not delve into the details of programming .NET Enterprise Services. An excellent reference for such information is Christian Nagel's "Enterprise Services with the .NET Framework" Spring.NET includes an example of using these classes, the 'calculator' example. More information can be found in the section, .NET Enterprise Services example.

26.3. Server Side

One of the main challenges for the exporting of a serviced component to the host is the need for them to be contained within a physical assembly on the file system in order to be registered with the COM+ Services. To make things more complicated, this assembly has to be strongly named before it can be successfully registered.

Spring provides two classes that allow all of this to happen.

- Spring.Enterprise.ServicedComponentExporter is responsible for exporting a single component and making sure that it derives from ServicedComponent class. It also allows you to specify class-level and method-level attributes for the component in order to define things such as transactional behavior, queuing, etc.
- Spring.Enterprise.EnterpriseServicesExporter corresponds to a COM+ application, and it allows you to specify list of components that should be included in the application, as well as the application name and other assembly-level attributes

Let's say that we have a simple service interface and implementation class, such as these:

```
namespace MyApp.Services
{
    public interface IUserManager
    {
        User GetUser(int userId);
        void SaveUser(User user);
    }

    public class SimpleUserManager : IUserManager
    {
        private IUserDao userDao;
        public IUserDao UserDao
        {
            get { return userDao; }
            set { userDao = value; }
        }
}
```

```
public User GetUser(int userId)
{
    return UserDao.FindUser(userId);
}

public void SaveUser(User user)
{
    if (user.IsValid)
    {
       UserDao.SaveUser(user);
    }
}
```

And the corresponding object definition for it in the application context config file:

Let's say that we want to expose user manager as a serviced component so we can leverage its support for transactions. First we need to export our service using the exporter ServicedComponentExporter as shown below

```
</dictionary>
</property>
</object>
```

The exporter defined above will create a composition proxy for our SimpleUserManager class that extends <code>ServicedComponent</code> and delegates method calls to SimpleUserManager instance. It will also adorn the proxy class with a <code>TransactionAtribute</code> and all methods with an <code>AutoCompleteAttribute</code>.

The next thing we need to do is configure an exporter for the COM+ application that will host our new component:

```
<object id="MyComponentExporter" type="Spring.Enterprise.Ent</pre>
    property name="ApplicationName" value="My COM+ Applicat
    cproperty name="Description" value="My enterprise service"
    property name="AccessControl">
        <object type="System.EnterpriseServices.ApplicationA</pre>
            property name="AccessChecksLevel" value="Applic"
        </object>
    </property>
    property name="Roles">
        st>
            <value>Admin : Administrator role
            <value>User : User role</value>
            <value>Manager : Administrator role</value>
        </list>
    </property>
    cproperty name="Components">
        st>
            <ref object="MyApp.EnterpriseServices.UserManage"</pre>
        </list>
    </property>
    cproperty name="Assembly" value="MyComPlusApp"/>
</object>
```

This exporter will put all proxy classes for the specified list of components into the specified assembly, sign the assembly, and register it with the specified COM+ application name. If application does not exist it will create it and configure it using values specified for Description, AccessControl and Roles properties.

26.4. Client Side

Because serviced component classes are dynamically generated and registered, you cannot instantiate them in your code using the new operator. Instead, you need to use

Spring.Enterprise.ServicedComponentFactory

definition, which also allows you to specify the configuration template for the component as well as the name of the remote server the component is running on, if necessary. An example is shown below

You can then inject this instance of the IUserManager into a client class and use it just like you would use original SimpleUserManager implementation. As you can see, by coding your services as plain .Net objects, against well defined service interfaces, you gain easy pluggability for your service implementation though this configuration, while keeping the core business logic in a technology agnostic PONO, i.e. Plain Ordinary .Net Object.

Chapter 27. Web Services

27.1. Introduction

While the out-of-the-box support for web services in .NET is excellent, there are a few areas that the Spring.NET thought could use some improvement. Spring adds the ability to perform dependency injection on standard asmx web services. Spring's .NET Web Services support also allows you to export a 'plain .NET object' as a .NET web service By "plain .NET object" we mean classes that do not contain infrastructure specific attributes, such as WebMethod. On the server side, Spring's .NET web service exporters will automatically create a proxy that adds web service attributes. On the client side you can use Spring IoC container to configure a client side proxy that you generated with standard command line tools. Additionally, Spring provides the functionality to create the web service proxy dynamically at runtime (much like running the command line tools but at runtime and without some of the tools quirks) and use dependency injection to configure the resulting proxy class. On both the server and client side, you can apply AOP advice to add behavior such as logging, exception handling, etc. that is not easily encapsulated within an inheritance hierarchy across the application.

27.2. Server-side

One thing that the Spring.NET team didn't like much is that we had to have all these .asmx files lying around when all said files did was specify which class to instantiate to handle web service requests.

Second, the Spring.NET team also wanted to be able to use the Spring.NET IoC container to inject dependencies into our web service instances. Typically, a web service will rely on other objects, service objects for example, so being able to configure which service object implementation to use is very useful.

Last, but not least, the Spring.NET team did not like the fact that creating a web service is an implementation task. Most (although not all) services are best implemented as normal classes that use coarse-grained service interfaces, and the decision as to whether a particular service should be exposed as a remote object, web service, or even an enterprise (COM+) component, should only be a matter of configuration, and not implementation.

An example using the web service exporter can be found in quickstart example named 'calculator'. More information can be found here 'Web Services example'.

27.2.1. Removing the need for .asmx files

Unlike web pages, which use . ASPX files to store presentation code, and codebehind classes for the logic, web services are completely implemented within the code-behind class. This means that .asmx files serve no useful purpose, and as such they should neither be necessary nor indeed required at all.

Spring.NET allows application developers to expose existing web services easily by registering a custom implementation of the WebServiceHandlerFactory class and by creating a standard Spring.NET object definition for the service.

By way of an example, consider the following web service...

```
namespace MyComany.MyApp.Services
{
    [WebService(Namespace="http://myCompany/services")]
    public class HelloWorldService
    {
        [WebMethod]
        public string HelloWorld()
```

```
return "Hello World!";
}
}
```

This is just a standard class that has methods decorated with the WebMethod attribute and (at the class-level) the WebService attribute. Application developers can create this web service within Visual Studio just like any other class.

All that one need to do in order to publish this web service is:

1. Register the Spring.Web.Services.WebServiceFactoryHandler as the HTTP handler for *.asmx requests within one's Web.config file.

Of course, one can register any other extension as well, but typically there is no need as Spring.NET's handler factory will behave exactly the same as a standard handler factory if said handler factory cannot find the object definition for the specified service name. In that case the handler factory will simply look for an .asmx file.

If you are using IIS7 the following configuration is needed

```
<system.webServer>
  <validation validateIntegratedModeConfiguration="false"/>
    <handlers>
        <add name="SpringWebServiceSupport" verb="*" path="*.asm
        </handlers>
  </system.webServer>
```

2. Create an object definition for one's web service.

```
<object name="HelloWorld" type="MyComany.MyApp.Services.Hell</pre>
```

Note that one is not absolutely required to make the web service object definition abstract (via the abstract="true" attribute), but this is a recommended best practice in order to avoid creating an unnecessary instance of the service. Because the .NET infrastructure creates instances of the target service object internally for each request, all Spring.NET needs to provide is the System.Type of the service class, which can be retrieved from the object definition even if it is marked as abstract.

That's pretty much it as we can access this web service using the value specified for the **name** attribute of the object definition as the service name:

```
http://localhost/MyWebApp/HelloWorld.asmx
```

27.2.2. Injecting dependencies into web services

For arguments sake, let's say that we want to change the implementation of the HelloWorld method to make the returned message configurable.

One way to do it would be to use some kind of message locator to retrieve an appropriate message, but that locator needs to implemented. Also, it would certainly be an odd architecture that used dependency injection throughout the application to configure objects, but that resorted to the service locator approach when dealing with web services.

Ideally, one should be able to define a property for the message within one's web service class and have Spring.NET inject the message value into it:

```
namespace MyApp.Services
{
    public interface IHelloWorld
    {
        string HelloWorld();
    }

    [WebService(Namespace="http://myCompany/services")]
    public class HelloWorldService : IHelloWorld
    {
        private string message;
    }
}
```

```
public string Message
{
    set { message = value; }
}

[WebMethod]
public string HelloWorld()
{
    return this.message;
}
}
```

The problem with standard Spring.NET DI usage in this case is that Spring.NET does not control the instantiation of the web service. This happens deep in the internals of the .NET framework, thus making it quite difficult to plug in the code that will perform the configuration.

The solution is to create a dynamic server-side proxy that will wrap the web service and configure it. That way, the .NET framework gets a reference to a proxy type from Spring.NET and instantiates it. The proxy then asks a Spring.NET application context for the actual web service instance that will process requests.

This proxying requires that one export the web service explicitly using the Spring.Web.Services.WebServiceExporter class; in the specific case of this example, one must also not forget to configure the Message property for said service:

The WebServiceExporter copies the existing web service and method attribute values to the proxy implementation (if indeed any are defined). Please

note however that existing values can be overridden by setting properties on the WebServiceExporter.



Interface Requirements

In order to support some advanced usage scenarios, such as the ability to expose an AOP proxy as a web service (allowing the addition of AOP advices to web service methods), Spring.NET requires those objects that need to be exported as web services to implement a (service) interface.

Only methods that belong to an interface will be exported by the WebServiceExporter.

27.2.3. Exposing PONOs as Web Services

Now that we are generating a server-side proxy for the service, there is really no need for it to have all the attributes that web services need to have, such as **WebMethod**. Because .NET infrastructure code never really sees the "real" service, those attributes are redundant as the proxy needs to have them on its methods, because that's what .NET deals with, but they are not necessary on the target service's methods.

This means that we can safely remove the WebService and WebMethod attribute declarations from the service implementation, and what we are left with is a plain old .NET object (a PONO). The example above would still work, because the proxy generator will automatically add WebMethod attributes to all methods of the exported interfaces.

However, that is still not the ideal solution. You would lose information that the optional WebService and WebMethod attributes provide, such as service namespace, description, transaction mode, etc. One way to keep those values is to leave them within the service class and the proxy generator will simply copy them to the proxy class instead of creating empty ones, but that really does defeat the purpose.

To add specific attributes to the exported web service, you can set all the necessary values within the definition of the service exporter, like so...

```
<object id="HelloWorldExporter" type="Spring.Web.Services.Web."</pre>
    roperty name="TargetName" value="HelloWorld"/>
    property name="Namespace" value="http://myCompany/servi
```

```
property name="Description" value="My exported HelloWor
    property name="MemberAttributes">
        <dictionary>
            <entry key="HelloWorld">
                <object type="System.Web.Services.WebMethodA</pre>
                     property name="Description" value="My s
                     property name="MessageName" value="Zdra
                </object>
            </entry>
        </dictionary>
    </property>
</object>
// or, once configuration improvements are implemented...
<web:service targetName="HelloWorld" namespace="http://myCon</pre>
    <description>My exported HelloWorld web service.</descri</pre>
    <methods>
        <method name="HelloWorld" messageName="ZdravoSvete">
            <description>My Spring-configured HelloWorld met
        </method>
    </methods>
</web:service>
```

Based on the configuration above, Spring.NET will generate a web service proxy for all the interfaces implemented by a target and add attributes as necessary. This accomplishes the same goal while at the same time moving web service metadata from implementation class to configuration, which allows one to export pretty much any class as a web service.

The WebServiceExporter also has a TypeAttributes IList property for applying attributes at the type level.



The attribute to confirms to the WSI basic profile 1.1 is not added by default. The in a future release. In the meantime use the TypeAttributes IList property to add [WebServiceBinding(ConformsTo=WsiProfiles.BasicP to the generated proxy.

One can also export only certain interfaces that a service class implements by setting the Interfaces property of the WebServiceExporter.

Distributed Objects Warning

Distributed Objects Warning

Just because you *can* export any object as a web service, doesn't mean that you *should*. Distributed computing principles still apply and you need to make sure that your services are not chatty and that arguments and return values are Serializable.

You still need to exercise common sense when deciding whether to use web services (or remoting in general) at all, or if local service objects are all you need.

27.2.4. Exporting an AOP Proxy as a Web Service

It is often useful to be able to export an AOP proxy as a web service. For example, consider the case where you have a service that is wrapped with an AOP proxy that you want to access both locally and remotely (as a web service). The local client would simply obtain a reference to an AOP proxy directly, but any remote client needs to obtain a reference to an exported web service proxy, that delegates calls to an AOP proxy, that in turn delegates them to a target object while applying any configured AOP advice.

Effecting this setup is actually fairly straightforward; because an AOP proxy is an object just like any other object, all you need to do is set the WebServiceExporter's TargetName property to the id (or indeed the name or alias) of the AOP proxy. The following code snippets show how to do this...

That's it as every call to the methods of the exported web service will be intercepted by the target AOP proxy, which in turn will apply the configured debugging and timing advice to it.

27.3. Client-side

On the client side, the main objection the Spring.NET team has is that client code becomes tied to a proxy *class*, and not to a service *interface*. Unless you make the proxy class implement the service interface manually, as described by Juval Lowy in his book "Programming .NET Components", application code will be less flexible and it becomes very difficult to plug in different service implementation in the case when one decides to use a new and improved web service implementation or a local service instead of a web service.

The goal for Spring.NET's web services support is to enable the easy generation of client-side proxies that implement a specific service interface.

27.3.1. Using VS.NET generated proxy

The problem with the web-service proxy classes that are generated by VS.NET or the WSDL command line utility is that they don't implement a service interface. This tightly couples client code with web services and makes it impossible to change the implementation at a later date without modifying and recompiling the client.

Spring.NET provides a simple <code>IFactoryObject</code> implementation that will generate a "proxy for proxy" (however obtuse that may sound). Basically, the <code>Spring.Web.Services.WebServiceProxyFactory</code> class will create a proxy for the VS.NET- / WSDL-generated proxy that implements a specified service interface (thus solving the problem with the web-service proxy classes mentioned in the preceding paragraph).

At this point, an example may well be more illustrative in conveying what is happening; consider the following interface definition that we wish to expose as a web service...

```
namespace MyCompany.Services
{
    public interface IHelloWorld
    {
       string HelloWorld();
    }
}
```

In order to be able to reference a web service endpoint through this interface, you need to add a definition similar to the example shown below to your client's application context:

What is important to notice is that the underlying implementation class for the web service does not have to implement the same <code>IHelloWorld</code> service interface... so long as matching methods with compliant signatures exist (a kind of duck typing), Spring.NET will be able to create a proxy and delegate method calls appropriately. If a matching method cannot be found, the Spring.NET infrastructure code will throw an exception.

That said, if you control both the client and the server it is probably a good idea to make sure that the web service class on the server implements the service interface, especially if you plan on exporting it using Spring.NET's WebServiceExporter, which requires an interface in order to work.

27.3.2. Generating proxies dynamically

The WebServiceProxyFactory can also dynamically generate a webservice proxy. The XML object definition for this factory object is shown below

One use-case where this proxy is very useful is when dealing with typed data sets through a web service. Leaving the pros and cons of this approach aside, the current behavior of the proxy generator in .NET is to create wrapper types for the typed dataset. This not only pollutes the solution with extraneous classes but also results in multiple wrapper types being created, one for each web service that uses the typed dataset. This can quickly get confusing. The proxy created by Spring allows you to reference you typed datasets directly, avoiding the above mentioned issues.

27.3.3. Configuring the proxy instance

The WebServiceProxyFactory also implements the interface, Spring.Objects.Factory.IConfigurableFactoryObjectlowing to specify configuration for the product that the WebServiceProxyFactory creates. This is done by specifying the ProductTemplate property. This is particularly useful for securing the web service. An example is shown below.

```
<object id="PublicarAltasWebService" type="Spring.Web.Servi</pre>
   cproperty name="ProxyType" value="My.WebService" />
   property name="ServiceInterface" value="My.IWebServiceI
   property name="ProductTemplate">
     <object>
       <!-- Configure the web service URL -->
       roperty name="Url" value="https://localhost/MyApp/
        <!-- Configure the Username and password for the we
       cproperty name="Credentials">
         <object type="System.Net.NetworkCredential, System</pre>
           property name="UserName" value="user"/>
           property name="Password" value="password"/>
         </object>
       </property>
       <!-- Configure client certificate for the web service
       property name="ClientCertificates">
         st>
           <object id="MyCertificate" type="System.Security</pre>
             <constructor-arg name="fileName" value="Certif"</pre>
             <constructor-arg name="password" value="notgoi</pre>
           </object>
         </list>
```

```
</property>
</object>
</object>
```

For an example of how using SOAP headers for authentication using the WebServiceExporter and WebServiceProxyFactory, refer to this <u>solution</u> on our wiki.

Chapter 28. Windows Communication Foundation (WCF)

28.1. Introduction

Spring's WCF support allows you to configure your WCF services via dependency injection and add additional behavior to them using Aspect-Oriented programming (AOP).

For those who would like to get their feet wet right way, check out the WcfQuickStart application in the examples directory.

28.2. Configuring WCF services via Dependency Injection

The technical approach used to perform dependency injection is based on dynamically creating an implementation of your service interface (a dynamic proxy) that retrieves a configured instance of your service type from the Spring container. This dynamic proxy is then the final service type that is hosted.



An alternative implementation approach that uses extensibility points in WCF to create and configure your WCF service was tried but proved to be limited in its approach was first taken (afaik) by Oran Dennison on his blog and several other The issue in using this approach is that if the service is configured to be a single [ServiceBehavior(InstanceContextMode=InstanceCon then the invocation of the IInstanceProvider is short-circuited. See the notes on the invocation of the IInstanceProvider is short-circuited. documentation here. While this would be the preferred approach, no acceptable v

28.2.1. Dependency Injection

In this approach you develop your WCF services as you would normally do. For example here is a sample service type taken from the quickstart example.

```
[ServiceContract(Namespace = "http://Spring.WcfQuickStar
public interface ICalculator
    [OperationContract]
    double Add(double n1, double n2);
    [OperationContract]
    double Subtract(double n1, double n2);
    [OperationContract]
    double Multiply(double n1, double n2);
    [OperationContract]
    double Divide(double n1, double n2);
    [OperationContract]
    string GetName();
}
```

The implementation for the methods is fairly obvious but an additional property, SleepInSeconds, is present. This is the property we will configure via dependency injection. Here is a partial listing of the implementation

```
public class CalculatorService : ICalculator
{
    private int sleepInSeconds;

    public int SleepInSeconds
    {
        get { return sleepInSeconds; }
        set { sleepInSeconds = value; }
}

    public double Add(double n1, double n2)
    {
        Thread.Sleep(sleepInSeconds*1000);
        return n1 + n2;
}

// additional implementation not shown for brevity
}
```

To configure this object with Spring, provide the XML configuration metadata as shown below as you would with any Spring managed object.



Note

The object must be declared as a 'prototype' object, i.e. not a singleton, in order to interact correctly with WCF instancing.

To host this service type in a standalone application define an instance of a Spring.ServiceModel.Activation.ServiceHostFacto and set is property TargetName to the id value of the previously defined service type. ServiceHostFactoryObject is a Spring IFactoryObject implementation. (See here for more information on IFactoryObjects and their interaction with the container.) The

ServiceHostFactoryObject will create an instance of Spring.ServiceModel.Activation.SpringServiceHos that will be the ServiceHost instance associated with your service type. This configuration for this step is shown below.

Additional service configuration can be done declaratively in the standard App.config file as shown below



It is important that the name of the service in the WCF declarative configuration section match the name of the Spring object definition

Spring.ServiceModel.Activation.SpringServiceHos is where the dynamic proxy for your service type is generated. This dynamic proxy will implement a single 'WCF' interface, the same on that your service type implements. The implementation of the service interface methods on the proxy will delegate to a wrapped 'target' object which is the object instance retrieved by name from the Spring container using the Spring API, ApplicationContext.GetObject(name). Since the object retrieved in this manner is fully configured, your WCF service is as well.

Outside of a standalone application you can also use the class Spring.ServiceModel.Activation.ServiceHostFacto (which inherits from

System.ServiceModel.Activation.ServiceHostFacto to host your services so that they can be configured via dependency injection. To use the dynamic proxy approached described here you should still refer to the name of the service as the name of the object definition used to configure the service type in the Spring container.

There are not many disadvantages to this approach other than the need to specify the service name as the name of the object definition in the Spring container and to ensure that singleton=false is used in the object definition. You can also use Spring.ServiceModel.Activation.ServiceHostFacto to host your service inside IIS but should still refer to the service by the name of the object in the Spring container.

28.3. Apply AOP advice to WCF services

In either approach to performing dependency injection you can apply additional AOP advice to your WCF services in the same way as you have always done in Spring. The following configuration shows how to apply some simple performance monitoring advice to all services in the Spring.WcfQuickStart namespace and is taken from the QuickStart example.

The aop:config section implicitly uses Spring's autoproxying features to add additional behavior to any objects defined in the container that match the pointcut criteria.

28.4. Creating client side proxies declaratively

To create a client side proxy based on the use of ChannelFactory<T>, you can use this rather ugly 'boiler plate' XML snippit that takes uses Spring's support for calling factory methods on object instances.



This will be shortened using a custom namespce in a future release

The value 'serverAppCalculatorEndpoint' refers to the name of an enpoints in the <cli>section of the standard WCF configuration inside of App.config.

28.5. Exporting PONOs as WCF Services

Much like the approach taken for .asmx web services Spring provides an [ServiceContract] will add that exporter [OperationContract] attributes by default to all public interface (PONO) class. The exporter methods а given Spring.ServiceModel.ServiceExporter and has various options to fine-tune what interfaces are exported and the specific attributes that get applied to each method and on that class. Here is a simple example

Spring does not provide any means to add [DataContract] [DataMember] attributes to method arguments of your service operations. As such, either you will do that yourself or you may choose to use a serializer other than DataContractSerializer, for example one that relies on method arguments that implement the ISerializable interface, having the [Serializable] attribute, or are serializable via the XmlSerializer. Use the latter serializers is a good way to migrate from an existing RCP based approach, such as using .NET remoting, to WCF in order to take advantage of the WCF runtime and avoid editing much existing code. You can then incrementally refactor and/or operations that create new use DataContractSerializer.

Spring Framework



Part V. Integration

This part of the reference documentation covers the Spring Framework's integration with a number of related enterprise .NET technologies.

- Chapter 29, Message Oriented Middleware Apache ActiveMQ
- Chapter 30, Message Oriented Middleware MSMQ
- Chapter 31, Scheduling and Thread Pooling

Chapter 29. Message Oriented Middleware - Apache ActiveMQ

29.1. Introduction

The goal of Spring's messaging is to increase your productivity when writing an enterprise strength messaging middleware applications. Spring achieves these goals in several ways. First it provides several helper classes that remove from the developer the incidental complexity and resource management issues that arise when using messaging APIs. Second, the design of these messaging helper classes promote best practices in designing a messaging application by promoting a clear separation between the messaging middleware specific code and business processing that is technology agnostic. This is generally referred to a "plain old .NET object" (or PONO) programming model.

This chapter discusses Spring's messaging support for providers whose API was modeled after the Java Message Service (JMS) API. Vendors who provide a JMS inspired API include Apache, TIBCO, IBM, and Progress Software. If you are using Microsoft's Message Queue, please refer to the specific MSMQ section.

As there is no de facto-standard common API across messaging vendors, Spring provides an implementation of its helper classes for each of the major messaging middleware vendors. The naming of the classes you will interact with most frequently will either be identical for each provider, but located in a different namespace, or have their prefix change to be the three-letter-acronym commonly associated with the message provider. The list of providers supported by Spring is show below along with their namespace and prefix.

- 1. Apache ActiveMQ (NMS) in namespace Spring.Messaging.Nms. 'Nms' is sometimes used as the class prefix
- 2. TIBCO EMS in namespace **Spring.Messaging.Ems**. 'Ems' is sometimes used as the class prefix (to becommerically available)
- 3. SonicMQ in namespace Spring.Messaging.Sonic, 'Jms' is sometimes used as the class prefix. (to be commercially available)
- 4. Websphere MQ in namespace Spring.Messaging.Xms, 'Xms' is sometimes used as the class prefix (to be commerically available)

JMS can be roughly divided into two areas of functionality, namely the production and consumption of messages. For message production and the synchronous consumption of messages the a template class, named

NmsTemplate, EmsTemplate (etc.) is used. Asynchronous message consumption is performed though a multi-threaded message listener container, SimpleMessageListenerContainer. This container is used to create Message-Driven PONOs (MDPs) which refer to a messaging callback class that consists of just 'plain .NET object's and is devoid of other artifacts. The any specific messaging types or **IMessageConverter** interface is used by both the template class and the message listener container to convert between provider message types and PONOs.

The namespace <code>Spring.Messaging.<Vendor>.Core</code> contains the messing template class (e.g. <code>NmsTemplate</code>). The template class simplifies the use of the messaging APIs by handling the creation and release of resources, much like the <code>AdoTemplate</code> does for ADO.NET. The JMS inspired APIs are low-level API, much like ADO.NET. As such, even the simplest of operations requires 10s of lines of code with the bulk of that code related to resource management of intermediate API objects Spring's messaging support, both in Java and .NET, addresses the error-prone boiler plate coding style one needs when using these APIs.

The design principle common to Spring template classes is to provide helper methods to perform common operations and for more sophisticated usage, delegate the essence of the processing task to user implemented callback interfaces. The messaging template follows the same design. The message template class offer various convenience methods for the sending of messages, consuming a message synchronously, and exposing the message Session and MessageProducer to the user.

The namespace Spring.Messaging.

<VendorAcronym>.Support.Converter provides a

IMessageConverter abstraction to convert between .NET objects and
messages. The namespace Spring.Messaging.

<VendorAcronym>.Support.Destinations provides various
strategies for managing destinations, such as providing a service locater for
destinations stored in a directory service.

Finally, the namespace Spring.Messaging. < VendorAcronym>.Connections provides an implementations of

the ConnectionFactory suitable for use in standalone applications.

The rest of the sections in this chapter discusses each of the major helper classes in detail. Please refer to the sample application that ships with Spring for additional hands-on usage.



To simplify documenting features that are common across all provider implementations of Spring's helper classes a specific provider, Apache ActiveMQ, was selected. As such when you see 'NmsTemplate' in the documentation, it also refers to EmsTemplate, XmsTemplate, etc. unless specifically documented otherwise. The provider specific API classes are typically named after their JMS counterparts with the possible exception of a leading 'I' in front of interfaces in order to follow .NET naming conventions. In the documentation these API artifacts are referred to as 'ConnectionFactory', 'Session', 'Message', etc. without the leading 'I'.

29.1.1. Separation of Concerns

The use of MessageConverters and a PONO programming model promote messaging best practices by applying the principal of Separation of Concerns to messaging based architectures. The infrastructure concern of publishing and consuming messages is separated from the concern of business processing. These two concerns are reflected in the architecture as two distinct layers, a message processing layer and a business processing layer. The benefit of this approach is that your business processing is decoupled from the messaging technology, making it more likely to survive technological changes over time and also easier to test. Spring's MessageConverters provides support for mapping messaging data types to PONOs. Aside from being the link between the two layers, MessageConverters provide a pluggable strategy to help support the evolution of a loosely coupled architecture over time. Message formats will change over time, typically by the addition of new fields. MessageConverters can be implemented to detect different versions of messages and perform the appropriate mapping logic to PONOs such so that multiple versions of a message can be supported simultaneously, a common requirement in enterprise messaging architectures.

29.1.2. Interoperability and provider portability

Messaging is a traditional area of Interoperability across heterogeneous systems with messaging vendors providing support on multiple operating systems (Windows, UNIX, Mainframes OS's) as well as multiple language bindings (C, C++, Java, .NET, Perl, etc.). In 199x the Java Community Process came up with a specification to provide a common API across messaging providers as well as define some common messaging functionality. This specification is know as the Java Message Service. From the API perspective, it can roughly be thought of as the messaging counterpart to the ADO.NET or JDBC APIs that provide portability across different database providers.

Given this history, when messaging vendors created their .NET APIs, many did so by creating their own JMS inspired API in .NET. There is no de facto-standard common API across messaging vendors. As such, portability across vendors using Spring's helper classes is done by changing the configuration schema in your configuration to a particular vendor and doing a 'search-and-replace' on the code base, changing the namespace and a few class names. While not ideal ,using Spring will push you in the direction of isolating the messaging specific classes in its own layer and therefore will reduce the impact of the changes you make to the code when switch providers. You business logic classes called into via Spring's messaging infrastructure will remain the same.

The NMS project from Apache addresses the lack of a common API across .NET messaging providers by providing an abstract interface based API for messaging and several implementations for different providers. At the time of this writing, the project is close to releasing a 1.0 version that supports ApacheMQ, MSMQ, and TIBCO EMS. There are a few outstanding issues at the moment that prevent one using NMS as a common API for all messaging providers but hopefully these issues will be resolved. Note, that NMS serves 'double' duty as the preferred API for messaging with ActiveMQ as well as a providing portability across different messaging providers.

29.1.3. The role of Messaging API in a 'WCF world'

Windows Communication Foundation (WCF) also supports message oriented middleware. Not surprisingly, a Microsoft Message Queuing (MSMQ) binding is provided as part of WCF. The WCF programming model is higher level than the traditional messaging APIs such as JMS and NMS since you are programing to a service interface and use metadata (either XML or attributes) to configure the messaging behavior. If you prefer to use this service-oriented, RPC style

approach, then look to see if a vendor provides a WCF binding for your messaging provider. Note that even with the option of using WCF, many people prefer to sit 'closer to the metal' when using messaging middleware, to access specific features and functionality not available in WCF, or simply because they are more comfortable with that programming model.

A WCF binding for Apache NMS is being developed as a separate project under the <u>Spring Extensions</u> umbrella project. Stay tuned for details.

29.2. Using Spring Messaging

29.2.1. Messaging Template overview

template classes (NmsTemplate, the messaging Code that uses EmsTemplate, etc) only needs to implement callback interfaces giving them a clearly defined contract. The IMessageCreator callback interface creates a message given a Session provided by the calling code in NmsTemplate. In order to allow for more complex usage of the provider messaging API, the callback ISessionCallback provides the user with the provider specific Session and the callback messaging IProducerCallback specific Session exposes a provider and MessageProducer pair.

Provider messaging APIs typically expose two types of send methods, one that takes delivery mode, priority, and time-to-live as quality of service (QOS) parameters and one that takes no QOS parameters which uses default values. Since there are many higher level send methods in NmsTemplate, the setting of the QOS parameters have been exposed as properties on the template class to avoid duplication in the number of send methods. Similarly, the timeout value for synchronous receive calls is set using the property ReceiveTimeout.



Instances of the NmsTemplate class are thread-safe once configured. This is important because it means that you can configure a single instance of a NmsTemplate and then safely inject this shared reference into multiple collaborators. To be clear, the NmsTemplate is stateful, in that it maintains a reference to a **ConnectionFactory**, but this state is not conversational state.

29.2.2. Connections

The NmsTemplate requires a reference to a ConnectionFactory. The ConnectionFactory serves as the entry point for working with the provider's messaging API. It is used by the client application as a factory to create connections to the messaging server and encapsulates various configuration parameters, many of which are vendor specific such as SSL configuration options.

29.2.3. Caching Messaging Resources

The standard API usage of NMS and other JMS inspired APIs involves creating many intermediate objects. To send a message the following 'API' walk is performed

IConnectionFactory->IConnection->ISession->IMessageProducer-

Between the ConnectionFactory and the Send operation there are three intermediate objects that are created and destroyed. To optimise the resource usage and increase performance two implementations of IConnectionFactory are provided.

29.2.3.1. SingleConnectionFactory

Spring. Messaging. Nms. Connections. SingleConnection will return the same connection on all calls to CreateConnection and ignore calls to Close.

29.2.3.2. CachingConnectionFactory

Spring.Messaging.Nms.Connections.CachingConnect extends the functionality of SingleConnectionFactory and adds the caching of Sessions, MessageProducers, and MessageConsumers.

The initial cache size is set to 1, use the property SessionCacheSize to increase the number of cached sessions. Note that the number of actual cached sessions will be more than that number as sessions are cached based on their acknowledgment mode, so there can be up to 4 cached session instances when SessionCacheSize is set to one, one for each AcknowledgementMode. MessageProducers and MessageConsumers are cached within their owning session and also take into account the unique properties of the producers and consumers when caching.

MessageProducers are cached based on their destination. MessageConsumers are cached based on a key composed of the destination, selector, noLocal delivery flag, and the durable subscription name (if

creating durable consumers).

29.2.4. Destination Management

In Java implementations of JMS, Connections and Destinations are 'administered objects' accessible though JNDI - a directory service much like ActiveDirectory. In .NET each vendor has selected a different approach to destination management. Some are JNDI inspired, allowing you to retrieve Connections and Destinations that were configured administratively. You can use these vendor specific APIs to perform dependency injection on references to JMS Destination objects in Spring's XML configuration file by creating am implementation of IObjectFactory or alternatively configuring the specific concrete class implementation for a messaging provider.

However, this approach of administered objects can be quite cumbersome if there are a large number of destinations in the application or if there are advanced destination management features unique to the messaging provider. Examples of such advanced destination management would be the creation of dynamic destinations or support for a hierarchical namespace of destinations. The NmsTemplate delegates the resolution of a destination name to a destination object by delegating to an implementation of the interface IDestinationResolver. DynamicDestinationResolver is the default implementation used by NmsTemplate and accommodates resolving dynamic destinations.

Quite often the destinations used in a messaging application are only known at runtime and therefore cannot be administratively created when the application is deployed. This is often because there is shared application logic between interacting system components that create destinations at runtime according to a well-known naming convention. Even though the creation of dynamic destinations are not part of the original JMS specification, most vendors have provided this functionality. Dynamic destinations are created with a name defined by the user which differentiates them from temporary destinations and are often not registered in a directory service. The API used to create dynamic destinations varies from provider to provider since the properties associated with the destination are vendor specific. However, a simple implementation choice that is sometimes made by vendors is to use the TopicSession method CreateTopic(string topicName) or the QueueSession

method CreateQueue(string queueName) to create a new destination with default destination properties. Depending on the vendor implementation, DynamicDestinationResolver may then also create a physical destination instead of only resolving one.

The boolean property PubSubDomain is used to configure the NmsTemplate with knowledge of what messaging 'domain' is being used. By default the value of this property is false, indicating that the point-to-point domain, Queues, will be used. This property is infrequently used as the provider messaging APIs are now largely agnostic as to which messaging 'domain' is used, referring to 'Destinations' rather than 'Queues' or 'Topics'. However, this property does influence the behavior of dynamic destination resolution via implementations of the IDestinationResolver interface.

You can also configure the NmsTemplate with a default destination via the property **DefaultDestination**. The default destination will be used with send and receive operations that do not refer to a specific destination.

29.2.5. Message Listener Containers

One of the most common uses of JMS is to concurrently process messages delivered asynchronously. A message listener container is used to receive messages from a message queue and drive the IMessageListener that is injected into it. The listener container is responsible for all threading of message reception and dispatches into the listener for processing. A message listener container is the intermediary between an Message-Driven PONO (MDP) and a messaging provider, and takes care of registering to receive messages, resource acquisition and release, exception conversion and suchlike. This allows you as an application developer to write the (possibly complex) business logic associated with receiving a message (and possibly responding to it), and delegates boilerplate messaging infrastructure concerns to the framework.

A subclass of AbstractMessageListenerContainer is used to receive messages from JMS and drive the Message-Driven PONOs (MDPs) that are injected into it. There are one subclasses of AbstractMessageListenerContainer packaged with Spring - SimpleMessageListenerContainer. Additional subclasses, in particular to participate in distributed transactions (if the provider supports it), will be provided in future releases. SimpleMessageListenerContainer creates a

fixed number of JMS sessions at startup and uses them throughout the lifespan of the container.

29.2.6. Transaction Management

Spring provides a NmsTransactionManager that manages transactions for a single ConnectionFactory. This allows messaging applications to leverage the managed transaction features of Spring as described in Chapter 17, Transactionmanagement. The NmsTransactionManager performs local resource transactions, binding a Connection/Session pair from the specified ConnectionFactory to the thread. NmsTemplate automatically detects such transactional resources and operates on them accordingly.

Using Spring's SingleConnectionFactory will result in a shared Connection, with each transaction having its own independent Session.

29.3. Sending a Message

The NmsTemplate contains three convenience methods to send a message. The methods are listed below.

- void Send(IDestination destination, IMessageCreator messageCreator)
- void Send(string destinationName, IMessageCreator messageCreator)
- void Send(IMessageCreator messageCreator)

The method differ in how the destination is specified. In first case the JMS Destination object is specified directly. The second case specifies the destination using a string that is then resolved to a messaging <code>Destination</code> object using the <code>IDestinationResolver</code> associated with the template. The last method sends the message to the destination specified by <code>NmsTemplate</code>'s <code>DefaultDestination</code> property.

All methods take as an argument an instance of IMessageCreator which defines the API contract for you to create the JMS message. The interface is show below

```
public interface IMessageCreator {
   IMessage CreateMessage(ISession session);
}
```

Intermediate Sessions and MessageProducers needed to send the message are managed by NmsTemplate. The session passed in to the method is never null. There is a similar set methods that use a delegate instead of the interface, which can be convenient when writing small implementation in .NET 2.0 using anonymous delegates. Larger, more complex implementations of the method 'CreateMessage' are better suited to an interface based implementation.

- void SendWithDelegate(IDestination destination, MessageCreatorDelegate messageCreatorDelegate)
- void SendWithDelegate(string

 void SendWithDelegate(MessageCreatorDelegate messageCreatorDelegate)

The declaration of the delegate is

```
public delegate IMessage MessageCreatorDelegate(ISession ses
```

The following class shows how to use the SendWithDelegate method with an anonymous delegate to create a MapMessage from the supplied Session object. The use of the anonymous delegate allows for very terse syntax and easy access to local variables. The NmsTemplate is constructed by passing a reference to a ConnectionFactory.

```
public class SimplePublisher
    private NmsTemplate template;
    public SimplePublisher()
        template = new NmsTemplate(new ConnectionFactory
    public void Publish(string ticker, double price)
        template.SendWithDelegate("APP.STOCK.MARKETDATA"
                      delegate(ISession session)
                       {
                           IMapMessage message = session.
                          message.Body.SetString("TICKER
                           message.Body.SetDouble("PRICE"
                           message.NMSPriority = 2;
                           return message;
                      });
    }
}
```

A zero argument constructor and ConnectionFactory property are also provided. Alternatively consider deriving from Spring's NmsGatewaySupport convenience base class which provides a ConnectionFactory property that will instantiate a NmsTemplate instance that is made available via the property NmsTemplate.

29.3.1. Using MessageConverters

In order to facilitate the sending of domain model objects, NmsTemplate has various send methods that take a .NET object as an argument for a message's data content. The overloaded methods ConvertAndSend and ReceiveAndConvert in NmsTemplate of the delegate the conversion process to an instance **IMessageConverter** interface. This interface defines a simple contract to convert between .NET objects and JMS messages. The default implementation SimpleMessageConverter supports conversion and TextMessage, byte[] BytesMesssage, between String and System.Collections.IDictionary and MapMessage. By using the converter, you and your application code can focus on the business object that is being sent or received via messaging and not be concerned with the details of how it is represented as a JMS message. There is also an XmlMessageConverter that converts objects to an XML string and vice-versa for sending via a TextMessage.

The family of ConvertAndSend messages are similar to that of the Send method with the additional argument of type IMessagePostProcessor. These methods are listed below.

- void ConvertAndSend(object message)
- void ConvertAndSend(object message, IMessagePostProcessor postProcessor)
- void ConvertAndSend(string destinationName, object message)
- void ConvertAndSend(string destinationName, object message, IMessagePostProcessor postProcessor);

- void ConvertAndSend(Destination destination, object message)
- void ConvertAndSend(Destination destination, object message, IMessagePostProcessor postProcessor)

The example below uses the default message converter to send a Hashtable as a message to the destination "APP.STOCK".

```
public void PublishUsingDict(string ticker, double price)
{
   IDictionary marketData = new Hashtable();
   marketData.Add("TICKER", ticker);
   marketData.Add("PRICE", price);
   template.ConvertAndSend("APP.STOCK.MARKETDATA", marketData
}
```

To accommodate the setting of message's properties, headers, and body that can not be generally encapsulated inside a converter class, the IMessageConverterPostProcessor interface gives you access to the message after it has been converted but before it is sent. The example below demonstrates how to modify a message header and a property after a Hashtable is converted to a message using the IMessagePostProcessor. The methods ConvertAndSendUsingDelegate allow for the use of a delegate to perform message post processing. This family of methods is listed below

- void ConvertAndSendWithDelegate(object message, MessagePostProcessorDelegate postProcessor)
- void ConvertAndSendWithDelegate(IDestination destination, object message, MessagePostProcessorDelegate postProcessor)
- void ConvertAndSendWithDelegate(string destinationName, object message, MessagePostProcessorDelegate postProcessor)

The declaration of the delegate is

public delegate IMessage MessagePostProcessorDelegate(IMessage)

The following code shows this in action.

29.4. Session and Producer Callback

While the send operations cover many common usage scenarios, there are cases when you want to perform multiple operations on a JMS Session or MessageProducer. The SessionCallback and ProducerCallback expose the Session and Session / MessageProducer pair respectfully. The Execute() methods on NmsTemplate execute these callback methods.

```
• public
           object
                     Execute(IProducerCallback
 action)
• public
            object
                       Execute(ProducerDelegate
 action)
• public
            object
                       Execute(ISessionCallback
 action)
• public
            object
                        Execute(SessionDelegate
 action)
```

Where ISessionCallback and IProducerCallback are

```
public interface IProducerCallback
{
    object DoInJms(Session session, MessageProducer producer
}
```

and

```
public interface ISessionCallback
{
    object DoInJms(Session session);
}
```

The delegate signatures are listed below and mirror the interface method signature

```
public delegate object SessionDelegate(ISession session);
public delegate object ProducerDelegate(ISession session, IN
```

29.5. Receiving a message

29.5.1. Synchronous Reception

While messaging middleware is typically associated with asynchronous processing, it is possible to consume messages synchronously. The overloaded Receive(..) methods on NmsTemplate provide this functionality. During a synchronous receive, the calling thread blocks until a message becomes available. This can be a dangerous operation since the calling thread can potentially be blocked indefinitely. The property ReceiveTimeout on NmsTemplate specifies how long the receiver should wait before giving up waiting for a message.

The Receive methods are listed below

- public Message Receive()
- public Message Receive(Destination destination)
- public Message Receive(string destinationName)
- public Message ReceiveSelected(string messageSelector)
- public Message ReceiveSelected(string destinationName, string messageSelector)
- public Message ReceiveSelected(Destination destination, string messageSelector)

The Receive method without arguments will use the DefaultDestination. The ReceiveSelected methods apply the provided message selector string to the MessageConsumer that is created.

The ReceiveAndConvert methods apply the template's message converter when receiving a message. The message converter to use is set using the property MessageConverter and is the SimpleMessageConverter implementation by default. These methods are listed below.

- public object ReceiveAndConvert()
- public object ReceiveAndConvert(Destination destination)
- public object ReceiveAndConvert(string destinationName)
- public object ReceiveSelectedAndConvert(string messageSelector)
- public object ReceiveSelectedAndConvert(string destinationName, string messageSelector)
- public object ReceiveSelectedAndConvert(Destination destination, string messageSelector)

29.5.2. Asynchronous Reception

Asynchronous reception of messages occurs by the messaging provider invoking a callback function. This is commonly an interface such as the IMessageListener interface shown below, taken from the TIBCO EMS provider.

```
public interface IMessageListener
{
    void OnMessage(Message message);
}
```

Other vendors may provide a delegate based version of this callback or even both a delegate and interface options. Apache ActiveMQ supports the use of delegates for message reception callbacks. As a programming convenience in Spring.Messaging.Nms.Core is an interface IMessageListener that can be used with NMS.

Below is a simple implementation of the IMessageListener interface that processes a message.

```
using Spring.Messaging.Nms.Core;
```

```
using Apache.NMS;
using Common.Logging;
namespace MyApp
{
 public class SimpleMessageListener : IMessageListener
        private static readonly ILog LOG = LogManager.GetLog
        private int messageCount;
        public int MessageCount
        {
            get { return messageCount; }
        }
        public void OnMessage(IMessage message)
            messageCount++;
            LOG.Debug("Message listener count = " + message(
            ITextMessage textMessage = message as ITextMessa
            if (textMessage != null)
            {
                LOG.Info("Message Text = " + textMessage.Tex
            } else
                LOG.Warn("Can not process message of type "
        }
}
```

Once you've implemented your message listener, it's time to create a message listener container.

You register you listener with a message listener container that specifies various messaging configuration parameters, such as the ConnectionFactory, and the number of concurrent consumers to create. There is an abstract base class for message listener containers,

AbstractMessageListenerContainer, and one concrete implementation, SimpleMessageListenerContainer. SimpleMessageListenerContainer creates a fixed number of

JMS Sessions/MessageConsumer pairs as set by the property ConcurrentConsumers. Here is a sample configuration

The above configuration will create 10 threads that process messages off of the queue named "APP.REQUEST". The threads are those owned by the messaging provider as a result of creating a MessageConsumer. Other important properties are ClientID, used to set the ClientID of the Connection and MessageSelector to specify the 'sql-like' message selector string. Durable subscriptions are supported via the properties SubscriptionDurable and DurableSubscriptionName. You may also register an exception listener using the property ExceptionListener.

A custom schema to create the **SimpleMessageListener** container is also provided. Using this schema the configuration above looks like the following

Exceptions that are thrown during message processing can be passed to an

implementation of IExceptionHandler and registered with the container via the property ExceptionListener. The registered IExceptionHandler will be invoked if the exception is of the type NMSException (or the equivalent root exception type for other providers). The SimpleMessageListenerContainer will logs the exception at error level and not propagate the exception to the provider. All handling of acknowledgement and/or transactions is done by the listener container. You can override the method HandleListenerException to change this behavior.

Please refer to the Spring SDK documentation for additional description of the features and properties of SimpleMessageListenerContainer.

29.5.3. The ISessionAwareMessageListener interface

The ISessionAwareMessageListener interface is a Spring-specific interface that provides a similar contract to the messaging provider's IMessageListener interface or Listener delegate/event, but also provides the message handling method with access to the Session from which the Message was received.

```
public interface ISessionAwareMessageListener
{
   void OnMessage(IMessage message, ISession session);
}
```

You can also choose to implement this interface and register it with the message listener container

29.5.4. MessageListenerAdapater

The MessageListenerAdapter class is the final component in Spring's asynchronous messaging support: in a nutshell, it allows you to expose almost any class to be invoked as a messaging callback (there are of course some constraints).

Consider the following interface definition. Notice that although the interface **IMessageListener** neither extends the nor ISessionAwareMessageListener interfaces, it can still be used as **PONOs** (MDP) of Message-Driven via the the а use

MessageListenerAdapter class. Notice also how the various message handling methods are strongly typed according to the contents of the various Message types that they can receive and handle.

```
public interface MessageHandler {
    void HandleMessage(string message);
    void HandleMessage(Hashtable message);
    void HandleMessage(byte[] message);
}
```

and a class that implements this interface...

```
public class DefaultMessageHandler : IMessageHandler {
    // stub implementations elided for bevity...
}
```

In particular, note how the above implementation of the IMessageHandler interface (the above DefaultMessageHandler class) has no messaging provider API dependencies at all. It truly is a PONO that we will make into an MDP via the following configuration.

The previous examples relies on the fact that the default IMessageConverter implementation of the MessageListenerAdapter is SimpleMessageConverter that can convert from messages to strings, byte[], and hashtables and object from a

ITextMessage, IBytesMessage, IMapMessage, and IObjectMessage respectfully. Below is an example of another MDP that can only handle the receiving of NMS ITextMessage messages. Notice how the message handling method is actually called 'Receive' (the name of the message handling method in a MessageListenerAdapter defaults to 'HandleMessage'), but it is configurable (as you will see below). Notice also how the 'Receive(..)' method is strongly typed to receive and respond only to NMS ITextMessage messages.

```
public interface TextMessageHandler {
    void Receive(ITextMessage message);
}
```

```
public class TextMessageHandler implements ITextMessageHandl
    // implementation elided for clarity...
}
```

The configuration of the attendant MessageListenerAdapter would look like this

Please note that if the above 'MessageListener' receives a Message of a type other than ITextMessage, a ListenerExecutionFailedException will be thrown (and subsequently handled by the container by logging the exception).

If your IMessageConverter implementation will return multiple object types, overloading the handler method is perfectly acceptable, the most specific matching method will be used. A method with an object signature would be

consider a 'catch-all' method of last resort. For example, you can have an handler interface as shown below.

```
public interface IMyHandler
{
    void DoWork(string text);
    void DoWork(OrderRequest orderRequest);
    void DoWork(InvoiceRequest invoiceRequest);
    void DoWork(object obj);
}
```

Another of the capabilities of the MessageListenerAdapter class is the ability to automatically send back a response Message if a handler method returns a nonvoid value. The adapter's message converter will be used to convert the methods return value to a message. The resulting message will then be sent to the Destination defined in the JMS Reply-To property of the original Message (if one exists), or the default Destination set on the MessageListenerAdapter (if one configured). Ιf Destination is found then has been no an InvalidDestinationException will be thrown (and please note that this exception will not be swallowed and will propagate up the call stack).

An interface that is typical when used with a message converter that supports multiple object types and has return values is shown below.

```
public interface IMyHandler
{
    string DoWork(string text);
    OrderResponse DoWork(OrderRequest orderRequest);
    InvoiceResponse DoWork(InvoiceRequest invoiceRequest);
    void DoWork(object obj);
}
```

29.5.5. Processing messages within a messaging transaction

Invoking a message listener within a transaction only requires reconfiguration of the listener container. Local message transactions can be activated by setting the property SessionAcknowledgeMode which for NMS is of the enum type AcknowledgementMode, to AcknowledgementMode. Transactional. Each

message listener invocation will then operate within an active messaging transaction, with message reception rolled back in case of listener execution failure.

Sending a response message (via ISessionAwareMessageListener) will be part of the same local transaction, but any other resource operations (such as database access) will operate independently. This usually requires duplicate message detection in the listener implementation, covering the case where database processing has committed but message processing failed to commit. See the discussion on the ActiveMQ web site here for more information combining local database and messaging transactions.

29.5.6. Messaging Namespace support

To use the NMS namespace elements you will need to reference the NMS schema. For information on how to set this up refer to Section A.2.6, "The nms messaging schema". The namespace consists of one top level elements: listener-container/> which can contain one or more listener/> child elements. Here is an example of a basic configuration for two listeners.

The example above is equivalent to creating two distinct listener container object definitions and two distinct MessageListenerAdapter object definitions as demonstrated in the section entitled <u>Section 29.5.4</u>, "MessageListenerAdapater". In addition to the attributes shown above, the listener element may contain several optional ones. The following table describes all available attributes:

Table 29.1. Attributes of the NMS listener > element Attribute Description

A object name for the hosting listener container. If not specified, a object name will be automatically generated.

destination (required)	The destination name for this listener, resolved through the IDestinationResolver strategy.
ref (required)	The object name of the handler object.
method	The name of the handler method to invoke. If the ref points to a IMessageListener or Spring ISessionAwareMessageListener, this attribute may be omitted.
response- destination	The name of the default response destination to send response messages to. This will be applied in case of a request message that does not carry a "NMSReplyTo" field. The type of this destination will be determined by the listener-container's "destination-type" attribute. Note: This only applies to a listener method with a return value, for which each result object will be converted into a response message.
subscription	The name of the durable subscription, if any.
selector	An optional message selector for this listener.
pubsub- domain	An optional boolean value. Set to true for the publish-subscribe domain (Topics) or false (the default) for point-to-point domain (Queues). This is useful when using the default implementation for destination resolvers.

The listener-container/> element also accepts several optional attributes. This allows for customization of the various strategies (for example, DestinationResolver) as well as basic messaging settings and resource references. Using these attributes, it is possible to define highly-customized listener containers while still benefiting from the convenience of the namespace.

The following table describes all available attributes. Consult the class-level SDK documentation of the AbstractMessageListenerContainer and its subclass

SimpleMessageListenerContainer for more detail on the individual properties.

Table 29.2. Attributes of the NMS tener-container> element

Attribute

Description

connection- factory	A reference to the NMS ConnectionFactory object (the default object name is 'ConnectionFactory').
destination- resolver	A reference to the IDestinationResolver strategy for resolving JMS Destinations.
message- converter	A reference to the IMessageConverter strategy for converting NMS Messages to listener method arguments. Default is a SimpleMessageConverter.
destination- type	The NMS destination type for this listener: queue, topic or durableTopic. The default is queue.
client-id	The NMS client id for this listener container. Needs to be specified when using durable subscriptions.
acknowledge	The native NMS acknowledge mode: auto, client, dups-ok or transacted. A value of transacted activates a locally transacted Session. As an alternative, specify the transaction-manager attribute described below. Default is auto.
concurrency	The number of concurrent sessions/consumers to start for each listener. Default is 1; keep concurrency limited to 1 in case of a topic listener or if queue ordering is important; consider raising it for general queues.
recovery- interval	The time interval between connection recovery attempts. The default is 5 seconds. Specify as a TimeSpan value using Spring's TimeSpanConverter (e.g. 10s, 10m, 3h, etc)
max- recovery- time	The maximum time try reconnection attempts. The default is 10 minutes. Specify as a TimeSpan value using Spring's TimeSpanConverter (e.g. 10s, 10m, 3h, etc)
auto-startup	Set whether to automatically start the listeners after initialization. Default is true, optionally set to false.

optionally set to false.

Chapter 30. Message Oriented Middleware - MSMQ

30.1. Introduction

The goals of Spring's MSMQ 3.0 messaging support is to raise the level of abstraction when writing MSMQ applications. The System.Messaging API is a low-level API that provides the basis for creating a messaging application. However, 'Out-of-the-box', System. Messaging leaves the act of creating sophisticated multi-threaded messaging servers and clients as an infrastructure activity for the developer. Spring fills this gap by proving easy to use helper classes that makes creating an enterprise messaging application easy. These helper classes take into account the nuances of System. Messaging API, such as its lack of thread-safety in many cases, the handling of so-called 'poison messages' (messages that are endlessly redelivered due to an unrecoverable exception during message processing), and combining database transactions with message transactions. Other goals of Spring's MSMQ messaging support are to support messaging best practices, in particular encouraging a clean architectural layering that separates the messaging middleware specifics from the core business processing.

Spring's approach to distributed computing has always been to promote a plain old .NET object approach or a PONO programming model. In this approach plain .NET objects are those that are devoid of any reference to a particular middleware technology. Spring provides the 'adapter' classes that converts between the middleware world, in this case MSMQ, and the oo-world of your business processing. This is done through the use of Spring's MessageListenerAdapter class and IMessageConverters.

The namespace Spring. Messaging provides the core functionality for messaging. It contains the class MessageQueueTemplate simplifies the use of System.Messaging.MessageQueue the lack of thread-safety handling in most of System.Messaging.MessageQueue's methods (for example Send). A single instance of MessageQueueTemplate can be used throughout your application and Spring will ensure that a different instance of a **MessageQueue** class used thread is per MessageQueueTemplate's methods. This per-thread instance of a System.Messaging.MessageQueue is also available via its property MessageQueue. The MessageQueueTemplate class is

also of either 'ambient' aware the presence of an System.Transaction's transaction local or a System. Messaging. MessageQueueTransaction. As such if you use MessageQueueTemplate's send and receive methods, unlike with plain use of System. Messaging. MessageQueue, you do not need to keep track of this information yourself and call the correct overloaded System. Messaging. MessageQueue method for specific When transaction environment. using System. Messaging. MessageQueueTransaction this would usually require you as a developer to come up with your own mechanism for passing around a MessageQueueTransaction to multiple classes and layers in your application. MessageQueueTemplate manages this for you, so you don't have to do so yourself. These resource management and transaction features of MessageQueueTemplate are quite analogous to the transactional features of Spring's AdoTemplate in case you are already familiar with that functionality.

For asynchronous reception Spring provides several multi-threaded message listener containers. You can pick and configure the container that matches your message transactional processing needs and configure poison-message handling policies. The message listener container leverages Spring's support for managing transactions. Both DTC, local messaging transactions, and local database transactions are supported. In particular, you can easily coordinate the commit and rollback of a local MessageQueueTransaction and a local database transaction when they are used together.

From a programming perspective, Spring's MSMQ support involves you configuring message listener containers and writing a callback function for message processing. On the sending side, it involves you learning how to use MessageQueueTemplate. In both cases you will quite likely want to take advantage of using MessageListenerConverters so you can better structure the translation from the System.Messaging.Message data structure to your business objects. After the initial learning hurdle, you should find that you will be much more productive leveraging Spring's helper classes to write enterprise MSMQ applications than rolling your own infrastructure. Feedback and new feature requests are always welcome.

The Spring.MsmqQuickstart application located in the examples directory of the

distribution shows this functionality in action.

30.2. A quick tour for the impatient

Here is a quick example of how to use Spring's MSMQ support to create a client that sends a message and a multi-threaded server application that receives the message. (The client code could also be used as-is in a multi-threaded environment but this is not demonstrated).

client On the side of you create an instance the MessageQueueTemplate class and configure it to use MessageQueue. This can be done programmatically but it is common to use dependency injection and Spring's XML configuration file to configure your client class as shown below.

The MessageQueue object is created via an instance of MessageQueueFactoryObject and the MessageQueueTemplate refers to this factory object by name and not by reference. The SimpleSender class looks like this

```
public class QuestionService : IQuestionService
{
  private MessageQueueTemplate messageQueueTemplate;

public MessageQueueTemplate {
  get { return messageQueueTemplate; }
  set { messageQueueTemplate = value; }
```

```
public void SendQuestion(string question)
{
   MessageQueueTemplate.ConvertAndSend(question);
}
```

This class can be shared across multiple threads and the MessageQueueTemplate will take care of managing thread local access to a System.Messaging.MessageQueue as well as any System.Messaging.IMessageFormatter instances.

Furthermore, since this is a transactional queue (only the name gives it away), the message will be sent using a single local messaging transaction. The conversion from the string to the underling message is managed by an instance of the IMessageConverter class. By default an implementation that XmlMessageFormatter with TargetType a System.String is used. You can configure the MessageQueueTemplate to use other IMessageConveter implementations that do conversions above and beyond what the 'stock' IMessageFormatters do. See the section on MessageConverters for more details.

On the receiving side we would like to consume the messages transactionally from the queue. Since no other database operations are being performed in our side select the server processing, we TransactionMessageListenerContainer and configure it to MessageQueueTransactionManager. use The MessageQueueTransactionManager an implementation Spring's IPlatformTransactionManager abstraction that provides a uniform API on top of various transaction manager (ADO.NET,NHibernate, MSMQ, etc). Spring's MessageQueueTransactionManager is for createing, committing, responsible and rolling back **MSMQ** MessageQueueTransaction.

While you can create the message listener container programmatically, we will show the declarative configuration approach below

```
<!-- Queue to receive from -->
<object id='questionTxQueue' type='Spring.Messaging.Suppor</pre>
  property name='Path' value='.\Private$\questionTxQueue'
  property name='MessageReadPropertyFilterSetAll' value='
</object>
<!-- MSMQ Transaction Manager -->
<object id="messageQueueTransactionManager" type="Spring.N</pre>
<!-- Message Listener Container that uses MSMQ transaction
<object id="transactionalMessageListenerContainer" type="$</pre>
  roperty name="MessageQueueObjectName" value="questionT
 roperty name="PlatformTransactionManager" ref="message"
  property name="MaxConcurrentListeners" value="10"/>
  property name="MessageListener" ref="messageListenerAda"
</object>
<!-- Adapter to call a PONO as a messaging callback -->
<object id="messageListenerAdapter" type="Spring.Messaging"</pre>
  cproperty name="HandlerObject" ref="guestionHandler"/>
</object>
<!-- The PONO class that you write -->
<object id="questionHandler" type="MyNamespace.QuestionHan</pre>
```

We have specified the queue to listen, that we want to consume the messages transactionally, process messages from the queue using 10 threads, and that our plain object that will handle the business processing is of the type <code>QuestionHandler</code>. The only class you need to write, <code>QuestionHandler</code>, looks like

```
public class QuestionHandler : IQuestionHandler
{
   public void HandleObject(string question)
   {
      // perform message processing here

      Console.WriteLine("Received question: " + question);

      // use an instance of MessageQueueTemplate and have other
}
```

```
// partake in the same local message transaction used t
}
```

That is general idea. You write the sender class using MessageQueueTemplate and the consumer class which does not refer to any messaging specific class. The rest is configuration of Spring provided helper classes.

Note that if the <code>HandleObject</code> method has returned a string value a reply message would be sent to a response queue. The response queue would be taken from the Message's own <code>ResponseQueue</code> property or can be specified explicitly using MessageListenerAdapter's <code>DefaultResponseQueueName</code> property.

If an exception is thrown inside the QuestionHandler, then the MSMQ transaction is rolled back, putting the message back on the queue for redelivery. If the exception is not due to a transient error in the system, but a logical processing exception, then one would get endless redelivery of the message - clearly not a desirable situation. These messages are so called 'poison messages' and a strategy needs to be developed to deal with them. This is left as a development task if you when using the System.Messaging APIs but Spring provides a strategy for handling poison messages, both for DTC based message reception as well as for local messaging transactions.

In the last part this 'quick tour' we will configure the message listener container to handle poison messages. This is done by creating an instance of SendToQueueExceptionHandler and setting the property MaxRetry to be the number of exceptions or retry attempts we are willing to tolerate before taking corrective actions. In this case, the corrective action is to send the message to another queue. We can then create other message listener containers to read from those queues and handle the messages appropriately or perhaps you will avoid automated processing of these messages and take manual corrective actions.

In the event of an exception while processing the message, the message transaction will be rolled back (putting the message back on the queue questionTxQueue for redelivery). If the same message causes an exception in times then it will be processing sent transactionally errorQuestionTxQueue and the message transaction will commit (removing it from the queue questionTxQueue). You can also specify that certain exceptions should commit the transaction (remove from the queue) but this is not shown for informatio this functionality below more non SendToQueueExceptionHandler implements the interface IMessageTransactionExceptionHandler (discussed below) so you can write your own implementations should the provided ones not meet your needs.

That's the quick tour folks. Hopefully you got a general feel for how things work, what requires configuration, and what is the code you need to write. The following sections describe each of Spring's helper classes in more detail. The sample application that ships with Spring is also a good place to get started.

30.3. Using Spring MSMQ

30.3.1. MessageQueueTemplate

The MessageQueueTemplate is used for synchronously sending and receiving messages. A single instance can be shared across multiple threads, unlike the standard System.Messaging.MessageQueue class. (One less resource management issue to worry about!) A thread-local instance of MessageQueue available class MessageQueueTemplate's property MessageQueue. Α MessageQueueTemplate is created by passing a reference to the name of a MessageQueueFactoryObject, you can think of it as a friendly name for your MessagingQueue and the recipe of how to create an of it. See the following instance section on MessageQueueFactoryObject for more information.

The MessageQueueTemplate also provides several convenience methods for sending and receiving messages. A family of overloaded ConvertAndSend and ReceiveAndConvert methods allow you to send and receive an object. The default message queue to send and receive from MessageQueueTemplate's using the MessageQueueObjectName. The responsibility of converting the object to a Message and vice versa is given to the template's associated **IMessageConverter** implementation. This can be set using the property MessageConverter. implementation, The default XmlMessageConverter, uses an XmlMessageFormatter with TargetType System.String. Note its set to System. Messaging. I Message Formatter classes are also not thread safe, so MessageQueueTemplate ensures that thread-local instances of IMessageConverter are used (as they generally wrap IMessageFormatter's that are not thread-safe).

You can use the MessageQueueTemplate to send messages to other MessageQueues by specifying their queue 'object name', the name of the MessageQueueFactoryObject.

```
void ConvertAndSend(object obj);
void ConvertAndSend(object obj, MessagePostProcessorDelegate
void ConvertAndSend(string messageQueueObjectName, object me
void ConvertAndSend(string messageQueueObjectName, object ob
object ReceiveAndConvert();
object ReceiveAndConvert(string messageQueueObjectName);
```

The transactional settings of the underlying overloaded System.Messaging.MessageQueue Send method that are used are based on the following algorithm.

1. If the message queue is transactional and there is an ambient MessageQueueTransaction in thread local storage (put there via the use of Spring's MessageQueueTransactionManager or TransactionalMessageListenerContainer), the message will be sent transactionally using the MessageQueueTransaction object in thread local storage.



o Note

This lets you group together multiple messaging operations within the same transaction without having to explicitly pass around the MessageQueueTransaction object.

- 2. f the message queue is transactional but there is no ambient MessageQueueTransaction, then a single message transaction is created on each messaging operation. (MessageQueueTransactionType = Single).
- 3. If there is an ambient System.Transactions transaction then that transaction will be used (MessageQueueTransactionType = Automatic).
- 4. If the queue is not transactional, then a non-transactional send

(MessageQueueTransactionType = None) is used.

The delegate MessagePostProcessorDelegate has the following signature

```
public delegate Message MessagePostProcessorDelegate(Message
```

This lets you modify the message after it has been converted from and object to a message using the <code>IMessageConverter</code> but before it is sent. This is useful for setting <code>Message</code> properties (e.g. <code>CorrelationId</code>, <code>AppSpecific</code>, <code>TimeToReachQueue</code>). Using anonymous delegates in .NET 2.0 makes this a very succinct coding task. If you have elaborate properties that need to be set, perhaps creating a custom <code>IMessageConverter</code> would be appropriate.

Overloaded **Send** and **Receive** operations that use the algorithm listed above to set transactional delivery options are also available. These are listed below

```
Message Receive();
Message Receive(string messageQueueObjectName);
void Send(Message message);
void Send(string messageQueueObjectName, Message message);
void Send(MessageQueue messageQueue, Message message);
```

Note that in the last Send method that takes a MessageQueue instance, it is the callers responsibility to ensure that this instance is not accessed from multiple threads. This **Send** method is commonly used when getting the MessageQueue from the ResponseQueue property of a Message during an asynchronous receive process. The receive timeout of the Receive ReceiveTimeout operations is the property of set using MessageQueueTemplate. The default value is MessageQueue.InfiniteTimeout (which is actually ~3 months).

The XML configuration snippit for defining a MessageQueueTemplate is shown

in the previous section and also is located in the MSMQ quickstart application configuration file Messaging.xml

30.3.2. MessageQueueFactoryObject

The MessageQueueFactoryObject is responsible for creating MessageQueue instances. You configure the factory with some basic information, namely the constructor parameters you are familiar with already when creating a standard MessageQueue instance, and then setting MessageQueue properties, such a Label etc. Some configuration tasks of a MessageQueue involve calling methods, for example to set which properties of the message to read. These available as properties to set on the MessageQueueFactoryObject. An example declarative configuration is shown below

Whenever an object reference is made to 'testqueue' an new instance of the MessageQueue class is created. This Spring's so-called 'prototype' model, which differs from 'singleton' mode. In the singleton creation mode whenever an object reference is made to a 'testqueue' the same MessageQueue instance would be used. So that a new instance can be retrieved based on need, the message listener containers take as an argument the name of the MessageQueueFactoryObject and not a reference. (i.e. use of 'value' instead of 'ref' in the XML).



The MessageQueueFactoryObject class is an ideal candidate for use future. This will allow you to use VS.NET IntelliSense to configure this commo is shown below

```
<mq:messageQueue id="testqueue" path=".\Private$\testqueue"
  <mq:properties label="MyLabel"/>
  </mq:messageQueue>
```

30.3.3. MessageQueue and IMessageConverter resource management

MessageQueues and IMessageFormatters (commonly used in IMessageConverter implementations) are not thread-safe. For example, only the following methods on MessageQueue are thread-safe, BeginPeek, BeginReceive, EndPeek, EndReceive, GetAllMessages, Peek, and Receive.

To isolate the creation logic of these classes, the factory interface IMessageQueueFactory is used. The interface is shown below

```
public interface IMessageQueueFactory
{
    MessageQueue CreateMessageQueue(string messageQueueC
    IMessageConverter CreateMessageConverter(string mess
}
```

A provided implementation, DefaultMessageQueueFactory will create an instance of each class per-thread. It delegates the creation of the MessageQueue instance to the Spring container. The argument, messageConverterObjectName, must be the id/name of a MessageQueueFactoryObject defined in the Spring container.

DefaultMessageQueueFactory leverages Spring's local thread storage support so it will work correctly in stand alone and web applications.

You can use the DefaultMessageQueueFactory independent of the

rest of Spring's MSMQ support should you need only the functionality it offers. MessageQueueTemplate and the listener containers create an instance of DefaultMessageQueueFactory by default. Should you want to share the same instance across these two classes, or provide your own custom implementation, use the property MessageQueueFactory on either MessageQueueTemplate or the message listener classe.s

30.3.4. Message Listener Containers

One of the most common uses of MSMQ is to concurrently process messages delivered asynchronously. This support is provided in Spring by message listener containers. A message listener container is the intermediary between an IMessageListener and a MessageQueue. (Note, message listener containers are conceptually different than Spring's Inversion of Control container, though it integrates and leverages the IoC container.) The message listener container takes care of registering to receive messages, participating in transactions, resource acquisition and release, exception conversion and suchlike. This allows you as an application developer to write the (possibly complex) business logic associated with receiving a message (and possibly responding to it), and delegate boilerplate MSMQ infrastructure concerns to the framework.

A subclass of AbstractMessageListenerContainer is used to receive messages from a MessageQueue. Which subclass you pick depends on your transaction processing requirements. The following subclasses are available in the namespace Spring.Messaging.Listener

- NonTransactionalMessageListenerContainer does not surround the receive operation with a transaction
- TransactionalMessageListenerContainer surrounds the receive operation with local (non-DTC) based transaction(s).
- DistributedTxMessageListenerContainer surrounds the receive operation with a distributed (DTC) transaction

Each of these containers use an implementation in which is based on Peeking for messages on a MessageQueue. Peeking is the only resource efficient approach that can be used in order to have MessageQueue receipt in conjunction with transactions, either local MSMQ transactions, local ADO.NET

based transactions, or DTC transactions. Each container can specify the number of threads that will be created for processing messages after the Peek occurs via the property MaxConcurrentListeners. Each processing thread will continue to listen for messages up until the timeout value specified by ListenerTimeLimit or until there are no more messages on the queue (whichever comes first). The default value of ListenerTimeLimit is TimeSpan.Zero, meaning that only one attempt to receive a message from the queue will be performed by each listener thread. The current implementation uses the standard .NET thread pool. Future implementations will use a custom (and pluggable) thread pool.

30.3.4.1. NonTransactionalMessageListenerContainer

This container performs a Receive operation on the MessageQueue without any transactional settings. As such messages will not be redelivered if an exception is thrown during message processing. Exceptions during message processing can be handled via an implementation of the interface IExceptionHandler. This can be set via the property ExceptionHandler on the listener. The IExceptionHandler interface is shown below

```
public interface IExceptionHandler
{
    void OnException(Exception exception, Message message)
```

An example of configuring a NonTransactionalMessageListenerContainer with an IExceptionHandler is shown below

```
</property>
</object>
<!-- Queue to respond to -->
<object id='msmqTestResponseQueue' type='Spring.Messaging.</pre>
  cproperty name='Path' value='.\Private$\testresponsequed
  property name='MessageReadPropertyFilterSetAll' value='
 property name='ProductTemplate'>
   <object>
      </object>
  </property>
</object>
<!-- Listener container -->
<object id="nonTransactionalMessageListenerContainer" type</pre>
  roperty name="MessageQueueObjectName" value="msmqTest()
 cproperty name="MaxConcurrentListeners" value="2"/>
 property name="ListenerTimeLimit" value="20s"/> <!-- 2</pre>
 property name="MessageListener" ref="messageListenerAda
 property name="ExceptionHandler" ref="exceptionHandler"
</object>
<!-- Delegate to plain .NET object for message handling --
<object id="messageListenerAdapter" type="Spring.Messaging"</pre>
  property name="DefaultResponseQueueName" value="msmqTes
 cproperty name="HandlerObject" ref="simpleHandler"/>
</object>
<!-- Classes you need to write -->
<object id="simpleHandler" type="MyNamespace.SimpleHandler</pre>
<object id="exceptionHandler" type="MyNamespace.SimpleExce")</pre>
```

The SimpleHandler class would look something like this

```
public class SimpleHandler : ISimpleHandler
{
  public void HandleObject(string txt)
  {
    // perform message processing...
    Console.WriteLine("Received text: " + txt);
```

30.3.4.2. TransactionalMessageListenerContainer

This message listener container performs receive operations within the context This class requires of local transaction. an instance Spring's IPlatformTransactionManager, either AdoPlatformTransactionManager, HibernateTransactionManager, or MessageQueueTransactionManager.

If you specify a MessageQueueTransactionManager then a MessageQueueTransaction will be started before receiving the message and used as part of the container's receive operation. As with other IPlatformTransactionManager implementation's, the this of transactional resources (in an instance the case MessageQueueTransaction class) is bound to thread local storage. MessageQueueTemplate will look in thread-local storage and use this 'ambient' transaction if found for its send and receive operations. The message invoked and if listener is no exception occurs. then the MessageQueueTransactionManager will commit the MessageQueueTransaction.

The message listener implementation can call into service layer classes that are made transactional using standard Spring declarative transactional techniques. In case of exceptions in the service layer, the database operation will be rolled back (nothing new here). TransactionalMessageListenerContainer will call it's IMessageTransactionExceptionHandler implementation to MessageQueueTransaction determine if the should (removing the message from the queue) or rollback (leaving the message on the queue for redelivery).



The use of a transactional service layer in combination with a MessageQueueTransactionManager is a powerful combination that can be used to achieve "exactly one" transaction message processing with database operations. This requires a little extra programming effort and is a more efficient alternative than using distributed transactions which are commonly associated with this functionality since both the database and the message transaction commit or rollback together.

The additional programming logic needed to achieve this is to keep track of the Message. Id that has been processed successfully within the transactional service layer. This is needed as there may be a system failure (e.g. power goes off) between the 'inner' database commit and the 'outer' messaging commit, resulting in message redelivery. The transactional service layer needs logic to detect if incoming message was processed successfully. It can do this by checking the database for an indication of successful processing, perhaps by recording the Message. Id itself in a status table. If the transactional service layer determines that the message has already been processed, it can throw a specific exception for this case. The container's exception handler will recognize this exception type and vote to commit (remove from the queue) the 'outer' messaging transaction. Spring provides an exception handler with this functionality, see SendToQueueExceptionHandler described below.

An example of configuring the TransactionalMessageListenerContainer using a MessageQueueTransactionManager is shown below

```
property name='MessageReadPropertyFilterSetAll' value='
  property name='ProductTemplate'>
    <object>
      property name='Label' value='MyTestResponseQueueLab
    </object>
  </property>
</object>
<!-- Transaction Manager for MSMQ Messaging -->
<object id="messageQueueTransactionManager" type="Spring.N</pre>
<!-- The transaction message listener container -->
<object id="transactionalMessageListenerContainer" type="$</pre>
  property name="MessageQueueObjectName" value="msmgTest()
  roperty name="PlatformTransactionManager" ref="message"
 property name="MaxConcurrentListeners" value="5"/>
 property name="ListenerTimeLimitIn" value="20s"/>
 roperty name="MessageListener" ref="messageListenerAda
  cproperty name="MessageTransactionExceptionHandler" ref=
</object>
<!-- Delegate to plain .NET object for message handling --
<object id="messageListenerAdapter" type="Spring.Messaging"</pre>
 property name="DefaultResponseQueueName" value="msmqTes
 cproperty name="HandlerObject" ref="simpleHandler"/>
</object>
<!-- Poison message handling -->
<object id="messageTransactionExceptionHandler" type="Spri</pre>
 property name="MaxRetry" value="5"/>
  cproperty name="MessageQueueObjectName" value="testTxErr
</object>
<!-- Classes you need to write -->
<object id="simpleHandler" type="MyNamespace.SimpleHandler</pre>
```

If you specify either AdoPlatformTransactionManager or HibernateTransactionManager then a local database transaction will be started before the receiving the message. By default, the container will

also start a local MessageQueueTransaction after the local database transaction has started, but before the receiving the message. This MessageQueueTransaction will be used to receive the message. By default the MessageQueueTransaction will be bound to thread local storage so that any MessageQueueTemplate send or receive operations will participate transparently in the same MessageQueueTransaction. If you do not want this behavior set the property

ExposeContainerManagedMessageQueueTransaction to false.

In case of exceptions during IMessageListener processing when using either either AdoPlatformTransactionManager or HibernateTransactionManager the container's IMessageTransactionExceptionHandler will determine if the MessageQueueTransaction should commit (removing it from the queue) or rollback (placing it back on the queue for redelivery). The listener exception will always trigger a rollback in the 'outer' database transaction.

Poison message handing, that is, the endless redelivery of a message due to exceptions during processing, can be detected using implementations of the IMessageTransactionExceptionHandler. This interface is shown below

```
public interface IMessageTransactionExceptionHandler
{
    TransactionAction OnException(Exception exception, Messa
}
```

The return value is an enumeration with the values Commit and Rollback. A specific implementation is provided that will move the poison message to another queue after a maximum number of redelivery attempts. See SendToQueueExceptionHandler described below. You can set a specific implementation to by setting TransactionalMessageListenerContainer's property MessageTransactionExceptionHandler

The IMessageTransactionExceptionHandler implementation

SendToQueueExceptionHandler keeps track of the Message's Id property in memory with a count of how many times an exception has occurred. If that count is greater than the handler's MaxRetry count it will be sent to another queue using the provided MessageQueueTransaction. The queue to send the message to is specified via the property MessageQueueObjectName.

30.3.4.3. DistributedTxMessageListenerContainer

This message listener container performs receive operations within the context of distributed transaction. A distributed transaction is started before a message is received. The receive operation participates in this transaction using by specifying MessageQueueTransactionType = Automatic. The transaction that is started is automatically promoted to two-phase-commit to avoid the default behavior of transaction promotion since the only reason to use this container is to use two different resource managers (messaging and database typically).

The commit and rollback semantics are simple, if the message listener does not throw an exception the transaction is committed, otherwise it is rolled back.

Exceptions in message listener processing are handled by implementations of the IDistributedTransactionExceptionHandler interface. This interface is shown below

```
public interface IDistributedTransactionExceptionHandler
{
    bool IsPoisonMessage(Message message);

    void HandlePoisonMessage(Message poisonMessage);

    void OnException(Exception exception, Message message);
}
```

the IsPoisonMessage method determines whether the incoming message poison message. This before method is called the IMessageListener is invoked. The container will call HandlePoisonMessage is IsPoisonMessage returns true and will then commit the distributed transaction (removing the message from the queue. Typical implementations of HandlePoisonMessage will move the poison message to another queue (under the same distributed transaction receive The used the message). class SendToQueueDistributedTransactionExceptionHandle detects poison messages by tracking the Message Id property in memory with a count of how many times an exception has occurred. If that count is greater than the handler's MaxRetry count it will be sent to another queue. The queue to specified message property send the to is via the MessageQueueObjectName.

30.4. MessageConverters

30.4.1. Using MessageConverters

In order to facilitate the sending of business model objects, the MessageQueueTemplate has various send methods that take a .NET object as an argument for a message's data content. The overloaded methods ConvertAndSend and ReceiveAndConvert in MessageQueue delegate the conversion process to an instance of the IMessageConverter interface. This interface defines a simple contract to convert between .NET objects and JMS messages. The interface is shown below

```
public interface IMessageConverter : ICloneable
{
    Message ToMessage(object obj);
    object FromMessage(Message message);
}
```

There are a standard implementations provided the simply wrap existing IMessageFormatter implementations.

- XmlMessageConverter uses a XmlMessageFormatter.
- BinaryMessageConverter uses a BinaryMessageFormatter
- ActiveXMessageConverter uses a ActiveXMessageFormatter

The default implementation used in MessageQueueTemplate and the message listener containers is an instance of XmlMessageConverter configured with a TargetType to be System.String. You specify the types that the XmlMessageConverter can convert though either the array property TargetTypes or TargetTypeNames. Here is an example taken from the QuickStart application

You can specify other IMessageConverter implementations using the MessageConverterObjectName property on the MessageQueueTemplate and MessageListenerAdapter.

Other implementations provided are

• XmlDocumentConverter - loads and saves an XmlDocument to the message BodyStream. This lets you manipulate directly the XML data independent of type serialization issues. This is quite useful if you use XPath expressions to pick out the relevant information to construct your business objects.

Other potential implementations:

- RawBytesMessageConverter directly write raw bytes to the message stream, compress
- CompressedMessageConverter compresses the message payload
- EncryptedMessageConverter encrypt the message (standard MSMQ encryptiong has several limitations)
- SoapMessageConverter use soap formatting.

30.5. Interface based message processing

30.5.1.1. MessageListenerAdapater

The MessageListenerAdapter allows methods of a class that does not implement the IMessageListener interface to be invoked upon message delivery. Lets call this class the 'message handler' class. To achieve this goal the MessageListenerAdapter implements the standard IMessageListener interface to receive a message and then delegates the processing to the message handler class. Since the message handler class does not contain methods that refer to MSMQ artifacts such as Message, the MessageListenerAdapter uses a IMessageConverter to bridge the MSMQ and 'plain object' worlds. As a reminder, the default XmlMessageConverter used in MessageQueueTemplate and the message listener containers converts from Message to string. Once the incoming message is converted to an object (string for example) a method with the name 'HandleMessage' is invoked via reflection passing in the string as an argument.

Using the default configuration of XmlMessageConverter in the message listeners, a simple string based message handler would look like this.

```
public class MyHandler
{
    public void HandleMessage(string text)
    {
        ...
    }
}
```

The next example has a similar method signature but the name of the handler method name has been changed to "DoWork", by setting the adapter's property DefaultHandlerMethod.

```
public interface IMyHandler
{
    void DoWork(string text);
```

```
}
```

If your IMessageConverter implementation will return multiple object types, overloading the handler method is perfectly acceptable, the most specific matching method will be used. A method with an object signature would be consider a 'catch-all' method of last resort.

```
public interface IMyHandler
{
    void DoWork(string text);
    void DoWork(OrderRequest orderRequest);
    void DoWork(InvoiceRequest invoiceRequest);
    void DoWork(object obj);
}
```

Another of the capabilities of the MessageListenerAdapter class is the ability to automatically send back a response Message if a handler method returns a non-void value. Any non-null value that is returned from the execution of the handler method will (in the default configuration) be converted to a string. The resulting string will then be sent to the ResponseQueue defined in the Message's ResponseQueue property of the original DefaultResponseQueueName Message, MessageListenerAdapter (if one has been configured) will be used. ResponseQueue is found then an Spring MessagingException will be thrown. Please note that this exception will not be swallowed and will propagate up the call stack.

Here is an example of Handler signatures that have various return types.

```
public interface IMyHandler
{
    string DoWork(string text);
    OrderResponse DoWork(OrderRequest orderRequest);
    InvoiceResponse DoWork(InvoiceRequest invoiceRequest);
    void DoWork(object obj);
}
```

The following configuration shows how to hook up the adapter to process incoming MSMQ messages using the default message converter.

30.6. Comparison with using WCF

The goals of Spring's MSMQ messaging support are quite similar to those of WCF with its MSMQ related bindings, in as much as a WCF service contract is a PONO (minus the attributes if you really picky about what you call a PONO). Spring's messaging support can give you the programming convenience of dealing with PONO contracts for message receiving but does not (at the moment) provide a similar PONO contract for sending, instead relying on explicit use of the MessageQueueTemplate class. This feature exists - some question whether it should for messaging - in the Java version of the Spring framework, see JmsInvokerServiceExporter and JmsInvokerProxyFactoryBean.

The good news is that if and when it comes time to move from a Spring MSMQ solution to WCF, you will be in a great position as the PONO interface used for business processing when receiving in a Spring based MSMQ application can easily be adapted to a WCF environment. There may also be some features unique to MSMQ and/or Spring's MSMQ support that you may find appealing over WCF. Many messaging applications still need to be 'closer to the metal' and this is not possible using the WCF bindings, for example Peeking and Label, AppSpecific properties, multicast.. An interesting recent quote by Yoel Arnon (MSMQ guru) "With all the respect to WCF, System.Messaging is still the major programming model for MSMQ programmers, and is probably going to remain significant for the foreseeable future. The message-oriented programming model is different from the service-oriented model of WCF, and many real-world solutions would always prefer it."

Chapter 31. Scheduling and Thread Pooling

31.1. Introduction

The Spring Framework features integration classes for scheduling support. Currently, Spring supports the Quartz Scheduler (http://quartznet.sourceforge.net/). The scheduler is set up using a IFactoryObject with optional references to Trigger instances, respectively. Furthermore, a convenience class for the Quartz Scheduler is available that allows you to invoke a method of an existing target object.

31.2. Using the Quartz.NET Scheduler

Quartz uses Trigger, Job and JobDetail objects to realize scheduling of all kinds of jobs. For the basic concepts behind Quartz, have a look at http://quartznet.sourceforge.net/. For convenience purposes, Spring offers a couple of classes that simplify the usage of Quartz within Spring-based applications.

31.2.1. Using the JobDetailObject

JobDetail objects contain all information needed to run a job. The Spring Framework provides a JobDetailObject that makes the JobDetail easier to configure and with sensible defaults. Let's have a look at an example:

The job detail object has all information it needs to run the job (ExampleJob). The timeout is specified in the job data dictionary. The job data dictionary is available through the JobExecutionContext (passed to you at execution time), but the JobDetailObject also maps the properties from the job data map to properties of the actual job. So in this case, if the ExampleJob contains a property named Timeout, the JobDetailObject will automatically apply it:

```
namespace Example.Quartz;
public class ExampleJob extends QuartzJobObject {
  private int timeout;

  /// <summary>
  /// Setter called after the ExampleJob is instantiated
  /// with the value from the JobDetailObject (5)
```

```
/// </summary>
public int Timeout {
    set { timeout = value; };
}

protected override void ExecuteInternal(JobExecutionContex
    // do the actual work
}
```

All additional settings from the job detail object are of course available to you as well.

Note: Using the **name** and **group** properties, you can modify the name and the group of the job, respectively. By default, the name of the job matches the object name of the job detail object (in the example above, this is **Example Job**).

31.2.2. Using the MethodInvokingJobDetailFactoryObject

Often you just need to invoke a method on a specific object. Using the MethodInvokingJobDetailFactoryObject you can do exactly this:

The above example will result in the doIt method being called on the exampleBusinessObject method (see below):

```
public class ExampleBusinessObject {
  // properties and collaborators
  public void DoIt() {
     // do the actual work
  }
```

```
<object id="ExampleBusinessObject" type="Examples.BusinessObject"</pre>
```

Using the MethodInvokingJobDetailFactoryObject, you don't need to create one-line jobs that just invoke a method, and you only need to create the actual business object and wire up the detail object.

By default, Quartz Jobs are stateless, resulting in the possibility of jobs interfering with each other. If you specify two triggers for the same JobDetail, it might be possible that before the first job has finished, the second one will start. If JobDetail classes implement the Stateful interface, this won't happen. The second job will not start before the first one has finished. make To iobs resulting from MethodInvokingJobDetailFactoryObject non-concurrent, set the concurrent flag to false.

```
<object id="JobDetail" type="Spring.Scheduling.Quartz.Method</pre>
  property name="TargetObject" ref="ExampleBusinessObject"
  property name="TargetMethod" value="DoIt" />
  cproperty name="Concurrent" value="false" />
</object>
```



By default, jobs will run in a concurrent fashion.

Also note that when using MethodInvokingJobDetailFactoryObject you can't use database persistence for Jobs. See the class documentation for additional details.

31.2.3. Wiring up jobs using triggers the and SchedulerFactoryObject

We've created job details and jobs. We've also reviewed the convenience class that allows to you invoke a method on a specific object. Of course, we still need to schedule the jobs themselves. This is done using triggers and a SchedulerFactoryObject. Several triggers are available within Quartz. Spring offers two subclassed triggers with convenient defaults:

CronTriggerObject and SimpleTriggerObject

Triggers need to be scheduled. Spring offers a SchedulerFactoryObject that exposes triggers to be set as properties. SchedulerFactoryObject schedules the actual jobs with those triggers.

Find below a couple of examples:

Now we've set up two triggers, one running every 50 seconds with a starting delay of 10 seconds and one every morning at 6 AM. To finalize everything, we need to set up the SchedulerFactoryObject:

More properties are available for the SchedulerFactoryObjecct for

you to set, such as the calendars used by the job details, properties to customize Quartz with, etc. Have a look at the <u>SchedulerFactoryObject SDK docs</u> for more information.

Spring Framework



Part VI. VS.NET Integration

This part of the reference documentation covers the Spring Framework's integration with VS.NET

• Chapter 32, Visual Studio.NET Integration

Chapter 32. Visual Studio.NET Integration

32.1. XML Editing and Validation

Most of this section is well travelled territory for those familiar with editing XML files in their favorite XML editor. The XML configuration data that defines the objects that Spring will manage for you are validated against the Spring.NET XML Schema at runtime. The location of the XML configuration data to create an <code>IApplicationContext</code> can be any of the resource locations supported by Spring's <code>IResource</code> abstraction. (See Section 7.1, "Introduction" for more information.) To create an <code>IApplicationContext</code> using a "standalone" XML configuration file the custom configuration section in the standard .NET application configuration would read:

```
<spring>
  <context>
        <resource uri="file://objects.xml"/>
        </context>
  </spring>
```

editor The VS.NET 2005 XMLattribute the can use xsi:schemaLocation as a hint to associate the physical location of a schema file with the XML document being edited. VS.NET 2002/2003 do not recognize the xsi:schemaLocation element. If you reference the Spring.NET XML schema as shown below, you can get intellisense and validation support while editing a Spring configuration file in VS.NET 2005. In order to get this functionality in VS.NET 2002/2003 you will need to register the schema with VS.NET or include the schema as part of your application project.

```
<object id="..." type="...">
    ...
    </object>
    ...
</objects>
```

It is typically more convenient to install the schema in VS.NET, even for VS.NET 2005, as it makes the xml a little less verbose and you don't need to keep copying the XSD file for each project you create. For VS.NET 2003 the schema directory will be either

C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Packages\schemas\xml $for\ VS\ 2003$

or

C:\Program Files\Microsoft Visual Studio
.NET\Common7\Packages\schemas\xml for VS.NET 2002

The VS.NET 2005 directory for XML schemas is

C:\Program Files\Microsoft Visual Studio
8\Xml\Schemas

Spring's .xsd schemas are located in the directory doc/schema. In that directory is also a NAnt build file to help copy over the .xsd files to the appropriate VS.NET locations. To execute this script simply type 'nant' in the doc/schema directory.

Once you have registered the schema with VS.NET you can adding only the namespace declaration to the objects element,

Once registered, the namespace declaration alone is sufficient to get intellisense and validation of the configuration file from within VS.NET. Alternatively, you can select the .xsd file to use by setting the targetSchema property in the Property Sheet for the configuration file.

As shown in the section <u>Section 5.2.6</u>, "<u>Using the container</u>" Spring.NET supports using .NET's application configuration file as the location to store the object definitions that will be managed by the object factory.

In this case VS.NET 2002/2003 will still provide you with intellisense help but you will not be able to fully validate the document as the entire schema for App.config is not known. To be able to validate this document one would need to install the NET_Configuration_File_schema and an additional schema that incorporates the Spring and context section in addition to the Objects would need to be created.

Validating schema is a new feature in VS 2005 it is validating all the time while

you edit, you will see any errors that it finds in the Error List window.

Keep these trade offs in mind as you decide where to place the bulk of your configuration information. Conventional wisdom is do quick prototyping with App.config and use another IResource location, file or embedded assembly resource, for serious development.

32.2. Versions of XML Schema

The schema was updated from Spring 1.0.1 to 1.0.2 in order to support generics. The schema for version 1.0.1 is located under http://www.springframework.net/xsd/1.0.1/ The for the latest version will be schema always located under http://www.springframework.net/xsd/

32.3. Integrated API help

Spring provides API documentation that can be integrated within Visual Studio. There are two versions of the documentation, one for VS.NET 2002/2003 and the other for VS 2005. They differ only in the format applied, VS 2005 using the sexy new format. Enjoy!

Spring Framework



Part VII. Quickstart applications

This part of the reference documentation covers the quickstart applications included with Spring that demonstrate features in a code centric manner.

- Chapter 33, IoC Quickstarts
- Chapter 34, AOP Guide
- Chapter 35, Portable Service Abstraction Quick Start
- Chapter 36, Web Quickstarts
- Chapter 37, SpringAir Reference Application
- Chapter 38, Data Access QuickStart
- Chapter 39, Transactions QuickStart
- Chapter 40, NHibernate QuickStart
- Chapter 41, Quartz QuickStart
- Chapter 42, NMS QuickStart
- Chapter 43, MSMQ QuickStart
- Chapter 44, WCF QuickStart

Chapter 33. IoC Quickstarts

33.1. Introduction

This chapter includes a grab bag of quickstart examples for using the Spring.NET framework.

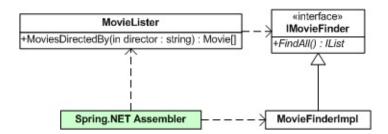
33.2. Movie Finder

The source material for this simple demonstration of Spring.NET's IoC features is lifted straight from Martin Fowler's article that discussed the ideas underpinning the IoC pattern. See <u>Inversion of Control Containers and the Dependency Injection pattern</u> for more information. The motivation for basing this quickstart example on said article is because the article is pretty widely known, and most people who are looking at IoC for the first time typically will have read the article (at the time of writing a <u>simple Google search for 'IoC'</u> yields the article in the first five results).

Fowler's article used the example of a search facility for movies to illustrate IoC and Dependency Injection (DI). The article described how a MovieLister object might receive a reference to an implementation of the IMovieFinder interface (using DI).

The IMOVieFinder returns a list of all movies and the MovieLister filters this list to return an array of Movieobjects that match a specified directors name. This example demonstrates how the Spring.NET IoC container can be used to supply an appropriate IMovieFinder implementation to an arbitrary MovieLister instance.

The C# code listings for the MovieFinder application can be found in the examples/Spring/Spring.Examples.MovieFinder directory off the top level directory of the Spring.NET distribution.



33.2.1. Getting Started - Movie Finder

The startup class for the MovieFinder example is the MovieApp class, which is an ordinary .NET class with a single application entry point...

```
using System;
namespace Spring.Examples.MovieFinder
{
   public class MovieApp
   {
      public static void Main ()
       {
       }
    }
}
```

What we want to do is get a reference to an instance of the MovieLister class... since this is a Spring.NET example we'll get this reference from Spring.NET's IoC container, the IApplicationContext. There are a number of ways to get a reference to an IApplicationContext instance, but for this example we'll be using an IApplicationContext that is instantiated from a custom configuration section in a standard .NET application config file...

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="spring">
            <section name="context" type="Spring.Context.Sur</pre>
            <section name="objects" type="Spring.Context.Sup"</pre>
        </sectionGroup>
    </configSections>
    <spring>
        <context>
            <resource uri="config://spring/objects"/>
        </context>
        <objects xmlns="http://www.springframework.net">
            <description>An example that demonstrates simpl
        </objects>
    </spring>
</configuration>
```

The objects that will be used in the example application will be configured as XML < object/> elements nested inside the < objects/> element.

The body of the Main method in the MovieApp class can now be fleshed out a little further...

```
using System;
using Spring.Context;
...
   public static void Main ()
   {
      IApplicationContext ctx = ContextRegistry.GetContext()
   }
...
```

As can be seen in the above C# snippet, a USing statement has been added to the MovieApp source. The Spring.Context namespace gives the application access to the IApplicationContext class that will serve as the primary means for the application to access the IoC container. The line of code...

```
IApplicationContext ctx = ContextRegistry.GetContext();
```

... retrieves a fully configured IApplicationContext implementation that has been configured using the named <Objects/> section from the application config file.

33.2.2. First Object Definition

As yet, no objects have been defined in the application config file, so let's do that now. The very miminal XML definition for the MOVieLister instance that we are going to use in the application can be seen in the following XML snippet...

```
<objects xmlns="http://www.springframework.net">
        <object name="MyMovieLister"
            type="Spring.Examples.MovieFinder.MovieLister, Spring.
        </object>
        </objects>
```

Notice that the full, assembly-qualified name of the MovieLister class has

been specified in the type attribute of the object definition, and that the definition has been assigned the (unique) id of MyMovieLister. Using this id, an instance of the object so defined can be retrieved from the IApplicationContext reference like so...

```
public static void Main ()
{
    IApplicationContext ctx = ContextRegistry.GetContext()
    MovieLister lister = (MovieLister) ctx.GetObject ("MyN")
}
...
```

The lister instance has not yet had an appropriate implementation of the IMOVieFinder interface injected into it. Attempting to use the MoviesDirectedBy method will most probably result in a nasty NullReferenceException since the lister instance does not yet have a reference to an IMOVieFinder. The XML configuration for the IMOVieFinder implementation that is going to be injected into the lister instance looks like this...

```
<objects xmlns="http://www.springframework.net">
        <object name="MyMovieFinder"
            type="Spring.Examples.MovieFinder.SimpleMovieFinder,
        </object>
</objects>
```

33.2.3. Setter Injection

What we want to do is inject the IMOVieFinder instance identified by the MyMoVieFinder id into the MoVieLister instance identified by the MyMoVieLister id, which can be accomplished using Setter Injection and the following XML...

```
<objects xmlns="http://www.springframework.net">
    <object name="MyMovieLister"
        type="Spring.Examples.MovieFinder.MovieLister, Spring.</pre>
```

When the MyMovieLister object is retrieved from (i.e. instantiated by) the IApplicationContext in the application, the Spring.NET IoC container will inject the reference to the MyMovieFinder object into the MovieFinder property of the MyMovieLister object. The MovieLister object that is referenced in the application is then fully configured and ready to be used in the application to do what is does best... list movies by director.

```
public static void Main ()
{
    IApplicationContext ctx = ContextRegistry.GetContext()
    MovieLister lister = (MovieLister) ctx.GetObject ("MyN Movie[] movies = lister.MoviesDirectedBy("Roberto Beni Console.WriteLine ("\nSearching for movie...\n");
    foreach (Movie movie in movies)
    {
        Console.WriteLine (
            string.Format ("Movie Title = '{0}', Director movie.Title, movie.Director));
    }
    Console.WriteLine ("\nMovieApp Done.\n\n");
}
```

To help ensure that the XML configuration of the MovieLister class must specify a value for the MovieFinder property, you can add the [Required] attribute to the MovieLister's MovieFinder property. The example code shows uses this attribute. For more information on using and configuring the [Required] attribute, refer to this <u>section</u> of the reference documentation.

33.2.4. Constructor Injection

Let's define another implementation of the **IMOVieFinder** interface in the application config file...

```
...
     <object name="AnotherMovieFinder"
          type="Spring.Examples.MovieFinder.ColonDelimitedMovieF
     </object>
...
```

This XML snippet describes an <code>IMovieFinder</code> implementation that uses a colon delimited text file as it's movie source. The C# source code for this class defines a single constructor that takes a <code>System.IO.FileInfo</code> as it's single constructor argument. As this object definition currently stands, attempting to get this object out of the <code>IApplicationContext</code> in the application with a line of code like so...

```
IMovieFinder = (IMovieFinder) ctx.GetObject ("Another
```

will result in a fatal Spring.Objects.Factory.ObjectCreationException, because the Spring.Examples.MovieFinder.ColonDelimitedMovie class does not have a default constructor that takes no arguments. If we want to use this implementation of the IMovieFinder interface, we will have to supply an appropriate constructor argument...

```
...
    <object name="AnotherMovieFinder"
        type="Spring.Examples.MovieFinder.ColonDelimitedMovieF
        <constructor-arg index="0" value="movies.txt"/>
        </object>
...
```

Unsurprisingly, the <constructor-arg/> element is used to supply constructor arguments to the constructors of managed objects. The Spring.NET IoC container uses the functionality offered by

System.ComponentModel.TypeConverter specializations to convert the Movies.txt string into an instance of the System.IO.FileInfo that is required by the single constructor of the Spring.Examples.MovieFinder.ColonDelimitedMovie (see Section 6.3, "Type conversion" for a more in depth treatment concerning the automatic type conversion functionality offered by Spring.NET).

So now we have two implementations of the IMOVieFinder interface that have been defined as distinct object definitions in the config file of the example application; if we wanted to, we could switch the implementation that the MyMovieLister object uses like so...

Note that there is no need to recompile the application to effect this change of implementation... simply changing the application config file and then restarting the application will result in the Spring.NET IoC container injecting the colon delimited implementation of the IMOVieFinder interface into the MyMovieLister object.

33.2.5. Summary

This example application is quite simple, and admittedly it doesn't do a whole lot. It does however demonstrate the basics of wiring together an object graph using an intuitive XML format. These simple features will get you through pretty much 80% of your object wiring needs. The remaining 20% of the available

configuration options are there to cover corner cases such as factory methods, lazy initialization, and suchlike (all of the configuration options are described in detail in the Chapter 5, The IoC container).

33.2.6. Logging

Often enough the first use of Spring.NET is also a first introduction to log4net. To kick start your understanding of log4net this section gives a quick overview. The authoritative place for information on log4net is the <u>log4net website</u>. Other good online tutorials are <u>Using log4net (OnDotNet article)</u> and <u>Quick and Dirty Guide to Configuring Log4Net For Web Applications</u>. Spring.NET is using version 1.2.9 whereas most of the documentation out there is for version 1.2.0. There have been some changes between the two so always double check at the log4net web site for definitive information. Also note that we are investigating using a "commons" logging library so that Spring.NET will not be explicity tied to log4net but will be able to use other logging packages such as NLog and Microsoft enterprise logging application block.

The general usage pattern for log4net is to configure your loggers, (either in App/Web.config or a seperate file), initialize log4net in your main application, declare some loggers in code, and then log log log. (Sing along...) We are using App.config to configure the loggers. As such, we declare the log4net configuration section handler as shown below

```
<section name="log4net" type="log4net.Config.Log4NetConfigur"</pre>
```

The corresponding configuration section looks like this

```
<!-- Set logging for Spring to INFO. Logger names in Spri
<logger name="Spring">
        <level value="INFO" />
        </logger>
</logder></logder>
```

The appender is the output sink - in this case the console. There are a large variety of output sinks such as files, databases, etc. Refer to the log4net <u>Config Examples</u> for more information. Of interest as well is the PatternLayout which defines exactly the information and format of what gets logged. Usually this is the date, thread, logging level, logger name, and then finally the log message. Refer to <u>PatternLayout Documentation</u> for information on how to customize.

The logging name is up to you to decide when you declare the logger in code. In the case of this example we used the convention of giving the logging name the name of the fully qualified class name.

```
private static readonly ILog LOG = LogManager.GetLogger(type
```

Other conventions are to give the same logger name across multiple classes that constitute a logical component or subsystem within the application, for example a data access layer. One tip in selecting the pattern layout is to shorten the logging name to only the last 2 parts of the fully qualified name to avoid the message sneaking off to the right too much (where can't see it) because of all the other information logged that precedes it. Shortening the logging name is done using the format %logger{2}.

To initialize the logging system add the following to the start of your application

```
XmlConfigurator.Configure();
```

Note that if you are using or reading information on version 1.2.0 this used to be called DOMConfigurator.Configure();

The logger sections associate logger names with logging levels and appenders. You have great flexibility to mix and match names, levels, and appenders. In this case we have defined the root logger (using the special tag root) to be at the debug level and have an console sink. We can then specialize other loggers with

different setting. In this case, loggers that start with "Spring" in their name are logged at the info level and also sent to the console. Setting the value of this logger from INFO to DEBUG will show you detailed logging information as the Spring container goes about its job of creating and configuring your objects. Coincidentally, the example code itself uses Spring in the logger name, so this logger also controls the output level you see from running MainApp. Finally, you are ready to use the simple logger api to log, i.e.

```
LOG.Info("Searching for movie...");
```

Logging exceptions is another common task, which can be done using the error level

```
try {
    //do work
{
    catch (Exception e)
{
    LOG.Error("Movie Finder is broken.", e);
}
```

33.3. ApplicationContext and IMessageSource

33.3.1. Introduction

The example program Spring.Examples.AppContext shows the use of the application context for text localization, retrieving objects contained in ResourceSets, and applying the values of embedded resource properties to an object. The values that are retrieved are displayed in a window.

The application context configuration file contains an object definition with the name <code>MessageSource</code> of the type <code>Spring.Context.Support.ResourceSetMessageSource</code> which implements the interface <code>IMessageSource</code>. This interface provides various methods for retrieving localized resources such as text and images as described in <code>Section 5.12.2</code>, "Using <code>IMessageSource</code>". When creating an instance of <code>IApplicationContext</code>, an object with the name 'messageSource' is searched for and used as the implementation for the context's <code>IMessageSource</code> functionality.

The ResourceSetMessageSource takes a list of ResourceManagers to define the collection of culture-specific resources. The ResourceManager can be contructed in two ways. The first way is to specifying a two part string consisting of the base resource name and the containing assembly. In this example there is an embedded resource file, Images.resx in the project. The helper factory class second is way to use ResourceManagerFactoryObject that takes a resource base name and assembly name as properties. This second way of specifying a ResourceManager is useful if you would like direct access to the ResourceManager in other parts of your application. In the example program an embedded resource file, MyResource.resx and a Spanish specific resource file, MyResources.es.resx are declared in this manner. The corresponding XML fragment is shown below

The main application creates the application context and then retrieves various resources via their key names. In the code all the key names are declared as static fields in the class ${\ensuremath{\mathsf{Keys}}}$. The resource file Images.resx contains image bubblechamber data under the key name (aka Keys.BUBBLECHAMBER). The Image image code (Image)ctx.GetResourceObject(Keys.BUBBLECHAMBER is used to retrieve the image from the context. The resource files MyResource.resx contains a text resource, Hello {0} {1} under the key name HelloMessage (aka Keys.HELLO_MESSAGE) that can be used for string text formatting purposes. The example code

retrieves the text string and replaces the placeholders in the string with the passed argument values resulting in the text, "Hello Mr. Anderson". The current culture is used to select the resource file MyResource.resx. If instead the Spanish culture is specified

Then the resource file MyResource.es.resx is used instead as in standard .NET

localization. Spring is simply delegating to .NET ResourceManager to select the appropriate localized resource. The Spanish version of the resource differs from the English one in that the text under the key HelloMessage is Hola {0} {1} resulting in the text "Hola Mr. Anderson".

As you can see in this example, the title "Mr." should not be used in the case of the spanish localization. The title can be abstracted out into a key of its own, called <code>FemaleGreeting</code> (aka Keys.FEMALE_GREETING). The replacement value for the message argument {0} can then be made localization aware by wrapping the key in a convenience class DefaultMessageResolvable. The code

will assign msg the value, Hello Mrs. Anderson, since the value for the key FemaleGreeting in MyResource.resx is 'Mrs.' Similarly, the code

will assign esMsg the value, Hola Senora Anderson, since the value for the key FemaleGreeting in MyResource.es.resx is 'Senora'.

Localization can also apply to objects and not just strings. The .NET 1.1 framework provides the utility class ComponentResourceManager that can apply multiple resource values to object properties in a performant manner. (VS.NET 2005 makes heavy use of this class in the code it generates for winform applications.) The example program has a simple class, Person, that has an integer property Age and a string property Name. The resource file, Person.resx contains key names that follow the pattern, person.<PropertyName>. In this case it contains person.Name and person.Age. The code to assign these resource values to an object is shown below

```
Person p = new Person();
ctx.ApplyResources(p, "person", CultureInfo.CurrentUICulture
```

While you could also use the Spring itself to set the properties of these objects, the configuration of simple properties using Spring will not take into account localization. It may be convenient to combine approaches and use Spring to configure the Person's object references while using IApplicationContext inside an AfterPropertiesSet callback (see IInitializingObject) to set the Person's culture aware properties.

33.4. ApplicationContext and IEventRegistry

33.4.1. Introduction

The example program Spring.Examples.EventRegistry shows how to use the application context to wire .NET events in a loosely coupled manner.

Loosely coupled eventing is normally associated with Message Oriented Middleware (MOM) where a daemon process acts as a message broker between other independent processes. Processes communicate indirectly with each other by sending messages though the message broker. The process that initiates the communication is known as a publisher and the process that receives the message is known as the subscriber. By using an API specific to the middleware these processes register themselves as either publishers or subscribers with the message broker. The communication between the publisher and subscriber is considered loosely coupled because neither the publisher nor subscriber has a direct reference to each other, the messages broker acts as an intermediary between the two processes. The IEventRegistry is the analogue of the message broker as applied to .NET events. Publishers are classes that invoke a .NET event, subscribers are the classes that register interest in these events, and the messages sent between them are instances of System. EventArgs. The implementation of IEventRegistry determines the exact semantics of the notification style and coupling between subscribers and publishers.

The IApplicationContext interface extends the IEventRegistry interface and implementations of IApplicationContext delegate the event registry functionality to an instance

Spring.Objects.Events.Support.EventRegistry.

IEventRegistry is a simple inteface with one publish method and two subscribe methods. Refer to <u>Section 5.12.4</u>, "<u>Loosely coupled events</u>" for a reminder of their signatures.

Spring.Objects.Events.Support.EventRegistry

implementation is essentially a convenience to decouple the event wiring process between publisher and subscribers. In this implementation, after the event wiring is finished, publishers are directly coupled to the subscribers via the standard .NET eventing mechanisms. Alternate implementations could increase the decoupling further by having the event registry subscribe to the events and be responsible for then notifying the subscribers.

In this example the class MyClientEventArgs is a subclass of System. EventArgs that defines a string property EventMessage. The class MyEventPublisher defines a public event with the delegate SimpleClientEvent(signature VOid object sender, MyClientEventArgs The void args method ClientMethodThatTriggersEvent1() fires this event. On the subscribing side, the class MyEventSubscriber contains a method, HandleClientEvents that matches the delegate signature and has a boolean property which is set to true if this method is called.

The publisher and subscriber classes are defined in an application context configuration file but that is not required in order to participate with the event registry. The main program, EventRegistryApp creates the application context and asks it for an instance of MyEventPublisher The publisher is registered with the event registry via the call, Ctx.PublishEvents(publisher). The event registry keeps a reference to this publisher for later use to register any subscribers that match its event signature. Two subscribers are then created and one of them is wired to the publisher by calling ctx.Subscribe(subscriber, the method typeof(MyEventPublisher)) Specifying the type indicates that the subscriber should be registered only to events from objects of the type MyEventPublisher. This acts as a simple filtering mechanism on the subscriber.

The publisher then fires the event using normal .NET eventing semantics and the subscriber is called. The subscriber prints a message to the console and sets a state variable to indicate it has been called. The program then simply prints the state variable of the two subscribers, showing that only one of them (the one that registered with the event registry) was called.

33.5. Pooling example

The idea is to build an executor backed by a pool of <code>QueuedExecutor</code>: this will show how Spring.NET provides some useful low-level/high-quality reusable threading and pooling abstractions. This executor will provide parallel executions (in our case <code>grep</code>-like file scans). Note: This example is not in the 1.0.0 release to its use of classes in the Spring.Threading namespace scheduled for release in Spring 1.1. To access the example please get the code from CVS (instructions) or from the download section of the Spring.NET website that contains an .zip with the full CVS tree.

Some information on QueuedExecutor is helpful to better understand the implementation and to possibly disagree with it. Keep in mind that the point is to show how to develop your own object-pool.

A QueuedExecutor is an executor where IRunnable instances are run serialy by a worker thread. When you Execute with a QueuedExecutor, your request is queued; at some point in the future your request will be taken and executed by the worker thread: in case of error the thread is terminated. However this executor recreates its worker thread as needed.

Last but not least, this executor can be shut down in a few different ways (please refer to the Spring.NET SDK documentation). Given its simplicity, it is very powerful.

Spring.Examples.Pool The provides example project an implementation of a pooled executor, backed by instances of Spring. Threading. QueuedExecutor: please ignore the fact that Spring. Threading includes already a very different implementation of PooledExecutor: here we wanto to pool of use a QueuedExecutors.

This executor will be used to implement a parallel recursive <code>grep</code>-like console executable.

33.5.1. Implementing Spring.Pool.IPoolableObjectFactory

In order to use the SimplePool implementation, the first thing to do is to

implement the IPoolableObjectFactory interface. This interface is intended to be implemented by objects that can create the type of objects that should be pooled. The SimplePool will call the lifecycle methods on IPoolableObjectFactory interface (MakeObject, ActivateObject, ValidateObject, PassivateObject, and DestroyObject) as appropriate when the pool is created, objects are borrowed and returned to the pool, and when the pool is destroyed.

In our case, as already said, we want to to implement a pool of **QueuedExecutor**. Ok, here the declaration:

```
public class QueuedExecutorPoolableFactory : IPoolableObject
{
```

the first task a factory should do is to create objects:

```
object IPoolableObjectFactory.MakeObject()
{
    // to actually make this work as a pooled executor
    // use a bounded queue of capacity 1.
    // If we don't do this one of the queued executors
    // will accept all the queued IRunnables as, by default
    // its queue is unbounded, and the PooledExecutor
    // will happen to always run only one thread ...
    return new QueuedExecutor(new BoundedBuffer(1));
}
```

and should be also able to destroy them:

```
void IPoolableObjectFactory.DestroyObject(object o)
{
    // ah, self documenting code:
    // Here you can see that we decided to let the
    // executor process all the currently queued tasks.
    QueuedExecutor executor = o as QueuedExecutor;
    executor.ShutdownAfterProcessingCurrentlyQueuedTasks();
}
```

When an object is taken from the pool, to satisfy a client request, may be the

object should be activated. We can possibly implement the activation like this:

```
void IPoolableObjectFactory.ActivateObject(object o)
{
    QueuedExecutor executor = o as QueuedExecutor;
    executor.Restart();
}
```

even if a QueuedExecutor restarts itself as needed and so a valid implementation could leave this method empty.

After activation, and before the pooled object can be successfully returned to the client, it is validated (should the object be invalid, it will be discarded: this can lead to an empty unusable pool ^[5]). Here we check that the worker thread exists:

```
bool IPoolableObjectFactory.ValidateObject(object o)
{
    QueuedExecutor executor = o as QueuedExecutor;
    return executor.Thread != null;
}
```

Passivation, symmetrical to activation, is the process a pooled object is subject to when the object is returned to the pool. In our case we simply do nothing:

```
void IPoolableObjectFactory.PassivateObject(object o)
{
}
```

At this point, creating a pool is simply a matter of creating an **SimplePool** as in:

```
pool = new SimplePool(new QueuedExecutorPoolableFactory(), s
```

33.5.2. Being smart using pooled objects

Taking advantage of the **using** keyword seems to be very important in these C# days, so we implement a very simple helper (PooledObjectHolder) that can allow us to do things like:

```
using (PooledObjectHolder holder = PooledObjectHolder.UseFro
{
    QueuedExecutor executor = (QueuedExecutor) holder.Pooled
    executor.Execute(runnable);
}
```

without worrying about obtaining and returning an object from/to the pool. Here is the implementation:

```
public class PooledObjectHolder : IDisposable
    IObjectPool pool;
    object pooled;
    /// <summary>
    /// Builds a new <see cref="PooledObjectHolder"/>
    /// trying to borrow an object form it
    /// </summary>
    /// <param name="pool"></param>
    private PooledObjectHolder(IObjectPool pool)
    {
        this.pool = pool;
        this.pooled = pool.BorrowObject();
    }
    /// <summary>
    /// Allow to access the borrowed pooled object
    /// </summary>
    public object Pooled
    {
        get
        {
            return pooled;
        }
    }
    /// <summary>
    /// Returns the borrowed object to the pool
    /// </summary>
    public void Dispose()
```

```
pool.ReturnObject(pooled);
}

/// <summary>
/// Creates a new <see cref="PooledObjectHolder"/> for t
/// given pool.
/// </summary>
public static PooledObjectHolder UseFrom(IObjectPool pool
{
    return new PooledObjectHolder(pool);
}
```

Please don't forget to destroy all the pooled istances once you have finished! How? Well using something like this in PooledQueuedExecutor:

```
public void Stop ()
{
    // waits for all the grep-task to have been queued ...
    foreach (ISync sync in syncs)
    {
        sync.Acquire();
    }
    pool.Close();
}
```

33.5.3. Using the executor to do a parallel grep

The use of the just built executor is quite straigtforward but a little tricky if we want to really exploit the pool.

```
private PooledQueuedExecutor executor;

public ParallelGrep(int size)
{
    executor = new PooledQueuedExecutor(size);
}

public void Recurse(string startPath, string filePattern, st
{
    foreach (string file in Directory.GetFiles(startPath, fi
```

```
{
    executor.Execute(new Grep(file, regexPattern));
}
foreach (string directory in Directory.GetDirectories(st
{
    Recurse(directory, filePattern, regexPattern);
}

public void Stop()
{
    executor.Stop();
}
```

```
public static void Main(string[] args)
    if (args.Length < 3)
    {
        Console.Out.WriteLine("usage: {0} regex directory fi
        Environment.Exit(1);
    }
    string regexPattern = args[0];
    string startPath = args[1];
    string filePattern = args[2];
    int size = 10;
    try
    {
        size = Int32.Parse(args[3]);
    catch
    {
    Console.Out.WriteLine ("pool size {0}", size);
    ParallelGrep grep = new ParallelGrep(size);
    grep.Recurse(startPath, filePattern, regexPattern);
    grep.Stop();
}
```

33.6. AOP

Refer to Chapter 34, AOP Guide.

[5] You may think that we can provide a smarter implementation and you are probably right. However, it is not so difficult to create a new pool in case the old one became unusable. It could not be your preferred choice but surely it leverages simplicity and object immutability

Chapter 34. AOP Guide

34.1. Introduction

This is an introductory guide to Aspect Oriented Programming (AOP) with Spring.NET.

This guide assumes little to no prior experience of having *used* Spring.NET AOP on the part of the reader. However, it *does* assume a certain familiarity with the terminology of AOP in general. It is probably better if you have read (or at least have skimmed through) the AOP section of the reference documentation beforehand, so that you are familiar with a) just what AOP is, b) what problems AOP is addressing, and c) what the AOP concepts of advice, pointcut, and joinpoint actually mean... this guide spends absolutely zero time defining those terms. Having said all that, if you are the kind of developer who learns best by example, then by all means follow along... you can always consult the reference documentation as the need arises (see Section 13.1.1, "AOP concepts").

The examples in this guide are **intentionally** simplistic. One of the core aims of this guide is to get you up and running with Spring.NET's flavor of AOP in as short a time as possible. Having to comprehend even a simple object model in order to understand the AOP examples would not be conducive to learning Spring.NET AOP. It is left as an exercise for the reader to take the concepts learned from this guide and apply them to his or her own code base. Again, having said all of that, this guide concludes with a number of cookbook-style AOP 'recipes' that illustrate the application of Spring.NET's AOP offering in a real world context; additionally, the Spring.NET reference application contains a number of Spring.NET AOP aspects particular to it's own domain model (see Chapter 37, SpringAir - Reference Application).

34.2. The basics

This initial section introduces the basics of defining and then applying some simple advice.

34.2.1. Applying advice

Lets see (a very basic) example of using Spring.NET AOP. The following example code simply applies advice that writes the details of an advised method call to the system console. Admittedly, this is not a particularly compelling or even useful application of AOP, but having worked through the example, you will then hopefully be able to see how to apply your own custom advice to perform useful work (transaction management, auditing, security enforcement, thread safety, etc).

Before looking at the AOP code proper lets quickly look at the domain classes that are the target of the advice (in Spring.NET AOP terminology, an instance of the following class is going to be the *advised object*.

```
public interface ICommand
{
    object Execute(object context);
}

public class ServiceCommand : ICommand
{
    public object Execute(object context)
    {
        Console.Out.WriteLine("Service implementation : [{0}}
        return null;
    }
}
```

Find below the advice that is going to be applied to the Object Execute(Object context) method of the ServiceCommand class. As you can see, this is an example of *around advice* (see Section 13.3.2, "Advice types").

```
public class ConsoleLoggingAroundAdvice : IMethodInterce
{
```

- Some simple code that merely prints out the fact that the advice is executing.
- The advised method is invoked.
- a The return value is captured in the returnValue variable.
- The value of the captured returnValue is printed out.
- **6** The previously captured returnValue is returned.

So thus far we have three artifacts: an interface (ICOMMand); an implementation of said interface (ServiceCommand); and some (trivial) advice (encapsulated by the ConsoleLoggingAroundAdvice class). All that remains is to actually apply the ConsoleLoggingAroundAdvice advice to the invocation of the Execute() method of the ServiceCommand class. Lets look at how to effect this programmatically...

```
ProxyFactory factory = new ProxyFactory(new ServiceComma
factory.AddAdvice(new ConsoleLoggingAroundAdvice());
ICommand command = (ICommand) factory.GetProxy();
command.Execute("This is the argument");
```

The result of executing the above snippet of code will look something like this...

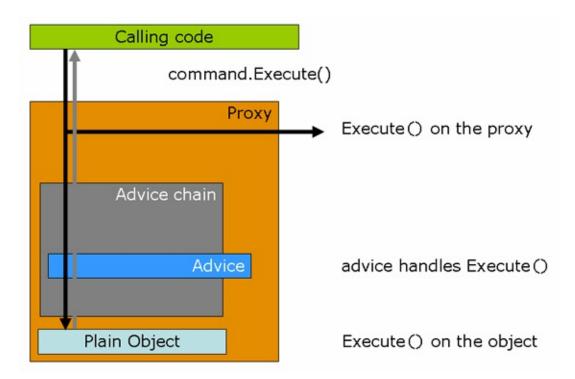
```
Advice executing; calling the advised method...
Service implementation : [This is the argument]
Advice executed; advised method returned
```

The output shows that the advice (the Console.Out statements from the ConsoleLoggingAroundAdvice was applied *around* the invocation

of the advised method.

So what is happening here? The fact that the preceding code used a class called ProxyFactory may have clued you in. The constructor for the ProxyFactory class took as an argument the object that we wanted to advise (in this case, an instance of the ServiceCommand class). We then added some advice (a ConsoleLoggingAroundAdvice instance) using the AddAdvice() method of the ProxyFactory instance. We then called the GetProxy() method of the ProxyFactory instance which gave us a proxy... an (AOP) proxy that proxied the target object (the ServiceCommand instance), and called the advice (a single instance of the ConsoleLoggingAroundAdvice in this case). When we invoked the Execute(object context) method of the proxy, the advice was 'applied' (executed), as can be seen from the attendant output.

The following image shows a graphical view of the flow of execution through a Spring.NET AOP proxy.



One thing to note here is that the AOP proxy that was returned from the call to the GetProxy() method of the ProxyFactory instance was cast to the

the ServiceCommand **ICommand** interface that target implemented. This is very important... currently, Spring.NET's AOP implementation mandates the use of an interface for advised objects. In short, this means that in order for your classes to leverage Spring.NET's AOP support, those classes that you wish to use with Spring.NET AOP must implement at least one interface. In practice this restriction is not as onerous as it sounds... in any case, it is *generally* good practice to program to interfaces anyway (support for applying advice to classes that do not implement any interfaces is planned for a future point release of Spring.NET AOP).

The remainder of this guide is concerned with fleshing out some of the finer details of Spring.NET AOP, but basically speaking, that's about it.

As a first example of fleshing out one of those finer details, find below some Spring.NET XML configuration that does *exactly* the same thing as the previous example; it should also be added that this declarative style approach to Spring.NET AOP is preferred to the programmatic style.

```
ICommand command = (ICommand) ctx["myServiceObject"];
command.Execute();
```

Some comments are warranted concerning the above XML configuration snippet. Firstly, note that the <code>ConsoleLoggingAroundAdvice</code> is itself a plain vanilla object, and is eligible for configuration just like any other class... if the advice itself needed to be injected with any dependencies, any such dependencies could be injected as normal.

Secondly, notice that the object definition corresponding to the object that is retrieved from the IoC container is a ProxyFactoryObject. The ProxyFactoryObject class is an implementation of the IFactoryObject interface; IFactoryObject implementations are treated specially by the Spring.NET IoC container... in this specific case, it is not a reference to the ProxyFactoryObject instance itself that is returned, but rather the object that the ProxyFactoryObject produces. In this case, it will be an advised instance of the ServiceCommand class.

Thirdly, notice that the target of the ProxyFactoryObject is an instance of the ServiceCommand class; this is the object that is going to be advised (i.e. invocations of its methods are going to be intercepted). This object instance is defined as an inner object definition... this is the preferred idiom for using the ProxyFactoryObject, as it means that other objects cannot acquire a reference to the raw object, but rather only the advised object.

Finally, notice that the advice that is to be applied to the target object is referred to by its object name in the list of the names of interceptors for the ProxyFactoryObject's interceptorNames property. In this particular case, there is only one instance of advice being applied... the ConsoleLoggingAroundAdvice defined in an object definition of the same name. The reason for using a list of object names as opposed to references to the advice objects themselves is explained in the reference documentation...

'... if the **ProxyFactoryObject**'s singleton property is set to false, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, an independent instance would need to be returned, so it is necessary to be able to obtain an instance of the prototype from the context; holding a reference isn't sufficient.'

34.2.2. Using Pointcuts - the basics

The advice that was applied in the previous section was rather indiscriminate with regard to which methods on the advised object were to be advised... the ConsoleLoggingAroundAdvice simply intercepted all methods (that were part of an interface implementation) on the target object.

This is great for simple examples and suchlike, but not so great when you only

want certain methods of an object to be advised. For example, you may only want those methods beginning with 'Start' to be advised; or you may only want those methods that are called with specific runtime argument values to be advised; or you may only want those methods that are decorated with a Lockable attribute to be advised.

The mechanism that Spring.NET AOP uses to discriminate about where advice is applied (i.e. which method invocations are intercepted) is encapsulated by the IPOintcut interface (see Section 13.2, "Pointcut API in Spring.NET"). Spring.NET provides many out-of-the-box implementations of the IPOintcut interface... the implementation that is used if none is explicitly supplied (as was the case with the first example) is the canonical TruePointcut: as the name suggests, this pointcut always matches, and hence all methods that can be advised will be advised.

So let's change the configuration of the advice such that it is only applied to methods that contain the letters 'Do'. We'll change the <code>ICommand</code> interface (and it's attendant implementation) to accommodate this...

```
public interface ICommand
{
    void Execute();
    void DoExecute();
}

public class ServiceCommand : ICommand
{
    public void Execute()
    {
        Console.Out.WriteLine("Service implementation :
      }

    public void DoExecute()
    {
        Console.Out.WriteLine("Service implementation :
      }
}
```

Please note that the advice itself (encapsulated within the

ConsoleLoggingAroundAdvice class) does not need to change; we are changing *where* this advice is applied, and not the advice itself.

Programmatic configuration of the advice, taking into account the fact that we only want methods that contain the letters 'Do' to be advised, looks like this...

```
ProxyFactory factory = new ProxyFactory(new ServiceComma
factory.AddAdvisor(new DefaultPointcutAdvisor(
    new SdkRegularExpressionMethodPointcut("Do"),
    new ConsoleLoggingAroundAdvice()));
ICommand command = (ICommand) factory.GetProxy();
command.DoExecute();
```

The result of executing the above snippet of code will look something like this...

```
Intercepted call: about to invoke next item in chain...
Service implementation...
Intercepted call: returned
```

The output indicates that the advice was applied around the invocation of the advised method, because the name of the method that was executed contained the letters 'Do'. Try changing the pertinent code snippet to invoke the Execute() method, like so...

```
ProxyFactory factory = new ProxyFactory(new ServiceComma
factory.AddAdvisor(
    new DefaultPointcutAdvisor(
    new SdkRegularExpressionMethodPointcut("Do"),
    new ConsoleLoggingAroundAdvice()));
ICommand command = (ICommand) factory.GetProxy();

// note that there is no 'Do' in this method name
command.Execute();
```

Run the code snippet again; you will see that the advice will not be applied : the pointcut is not matched (the method name does not contain the letters 'DO'), resulting in the following (unadvised) output...

```
Service implementation...
```

XML configuration that accomplishes exactly the same thing as the previous programmatic configuration example can be seen below...

```
<object id="consoleLoggingAroundAdvice"</pre>
    type="Spring.Aop.Support.RegularExpressionMethodPoir
    cproperty name="pattern" value="Do"/>
    property name="advice">
        <object type="Spring.Examples.AopQuickStart.Cons</pre>
    </property>
</object>
<object id="myServiceObject"</pre>
        type="Spring.Aop.Framework.ProxyFactoryObject">
    property name="target">
        <object id="myServiceObjectTarget"</pre>
            type="Spring.Examples.AopQuickStart.Service(
    </property>
    property name="interceptorNames">
        st>
            <value>consoleLoggingAroundAdvice</value>
        </list>
    </property>
</object>
```

You'll will perhaps have noticed that this treatment of pointcuts introduced the concept of an advisor (see <u>Section 13.4, "Advisor API in Spring.NET"</u>). An advisor is nothing more the composition of a pointcut (i.e. where advice is going to be applied), and the advice itself (i.e. what is going to happen at the interception point). The consoleLoggingAroundAdvice object defines an advisor that will apply the advice to all those methods of the advised object that match the pattern 'Do' (the pointcut). The pattern to match against is supplied pattern simple string value to the property the as RegularExpressionMethodPointcutAdvisor class.

34.3. Going deeper

The first section should (hopefully) have demonstrated the basics of firstly defining advice, and secondly, of choosing where to apply that advice using the notion of a pointcut. Of course, there is a great deal more to Spring.NET AOP than the aforementioned single advice type and pointcut. This section continues the exploration of Spring.NET AOP, and describes the various advice and pointcuts that are available for you to use (yes, there is more than one type of advice and pointcut).

34.3.1. Other types of Advice

The advice that was demonstrated and explained in the preceding section is what is termed 'around advice'. The name 'around advice' is used because the advice is applied around the target method invocation. In the specific case of the ConsoleLoggingAroundAdvice advice that was defined previously, the target was made available to the advice as an IMethodInvocation object... a call was made to the Console class before the target was invoked, and a call was made to the Console class after the target method invocation was invoked. The advice surrounded the target, one could even say that the advice was totally 'around' the target... hence the name, 'around advice'.

'around advice' provides one with the opportunity to do things both **before** the target gets a chance to do anything, and **after** the target has returned: one even gets a chance to inspect (and possibly even totally change) the return value.

Sometimes you don't need all that power though. If we stick with the example of the <code>ConsoleLoggingAroundAdvice</code> advice, what if one just wants to log the fact that a method was called? In that case one doesn't need to do anything <code>after</code> the target method invocation is to be invoked, nor do you need access to the return value of the target method invocation. In fact, you only want to do something <code>before</code> the target is to be invoked (in this case, print out a message to the system <code>Console</code> detailing the name of the method). In the tradition of good programming that says one should use only what one needs and no more, <code>Spring.NET</code> has another type of advice that one can use... if one only wants to do something <code>before</code> the target method invocation is invoked, why bother with having to manually call the <code>Proceed()</code> method? The most expedient solution simply is to use <code>'before advice'</code>.

34.3.1.1. Before advice

'before advice' is just that... it is advice that runs before the target method invocation is invoked. One does not get access to the target method invocation itself, and one cannot return a value... this is a good thing, because it means that you cannot inadvertently forget to call the Proceed() method on the target, and it also means that you cannot inadvertently forget to return the return value of the target method invocation. If you don't need to inspect or change the return value, or even do anything after the successful execution of the target method invocation, then 'before advice' is just what you need.

'before advice' in Spring.NET is defined by the IMethodBeforeAdvice interface in the Spring. Aop namespace. Lets just dive in with an example... we'll use the same scenario as before to keep things simple. Let's define the 'before advice' implementation first.

Let's apply a single instance of the ConsoleLoggingBeforeAdvice advice to the invocation of the Execute() method of the ServiceCommand. What follows is programmatic configuration; as you can see, its pretty much identical to the previous version... the only difference is that we're using our new 'before advice' (encapsulated as an instance of the ConsoleLoggingBeforeAdvice class).

```
ProxyFactory factory = new ProxyFactory(new ServiceComma
factory.AddAdvice(new ConsoleLoggingBeforeAdvice());
ICommand command = (ICommand) factory.GetProxy();
command.Execute();
```

The result of executing the above snippet of code will look something like this...

```
Intercepted call to this method : Execute
The target is : Spring.Examples.AopQui
The arguments are :
```

The output clearly indicates that the advice was applied **before** the invocation of the advised method. Notice that in contrast to 'around advice', with 'before advice' there is no chance of forgetting to call the Proceed() method on the target, because one does not have access to the IMethodInvocation (as is the case with 'around advice')... similarly, you cannot forget to return the return value either.

If you can use 'before advice', then do so. The simpler programming model offered by 'before advice' means that there is less to remember, and thus potentially less things to get wrong.

Here is the Spring.NET XML configuration for applying our *'before advice'* declaratively...

34.3.1.2. After advice

Just as 'before advice' defines advice that executes **before** an advised target, 'after advice' is advice that executes **after** a target has been executed.

'after advice' in Spring.NET is defined by the IAfterReturningAdvice interface in the Spring.Aop namespace. Again, lets just fire on ahead with an example... again, we'll use the same scenario as before to keep things simple.

```
public class ConsoleLoggingAfterAdvice : IAfterReturning
{
    public void AfterReturning(
        object returnValue, MethodInfo method, object[]
    {
        Console.Out.WriteLine("This method call returned
        Console.Out.WriteLine("
                                   The target was
        Console.Out.WriteLine("
                                   The arguments were
        if(args != null)
            foreach (object arg in args)
                Console.Out.WriteLine("\t: " + arg);
        Console.Out.WriteLine(" The return value is
    }
}
```

Let's apply a single instance of the ConsoleLoggingAfterAdvice advice to the invocation of the Execute() method of the ServiceCommand. What follows is programmatic configuration; as you can, its pretty much identical to the 'before advice' version (which in turn was pretty much identical to the original 'around advice' version)... the only real difference is that we're using our new 'after advice' (encapsulated as an instance of the ConsoleLoggingAfterAdvice class).

```
ProxyFactory factory = new ProxyFactory(new ServiceComma
factory.AddAdvice(new ConsoleLoggingAfterAdvice());
ICommand command = (ICommand) factory.GetProxy();
```

```
command.Execute();
```

The result of executing the above snippet of code will look something like this...

```
This method call returned successfully : Execute
```

The target was : Spring.Examples

The arguments were :

The return value is : null

The output clearly indicates that the advice was applied **after** the invocation of the advised method. Again, it bears repeating that your real world development will actually have an advice implementation that does something useful after the invocation of an advised method. Notice that in contrast to 'around advice', with 'after advice' there is no chance of forgetting to call the Proced() method on the target, because just like 'before advice' you don't have access to the IMethodInvocation... similarly, although you get access to the return value of the target, you cannot forget to return the return value either. You can however change the state of the return value, typically by setting some of its properties, or by calling methods on it.

The best-practice rule for 'after advice' is much the same as it is for 'before advice'; namely that if you can use 'after advice', then do so (in preference to using 'around advice'). The simpler programming model offered by 'after advice' means that there is less to remember, and thus less things to get potentially wrong.

A possible use case for 'after advice' would include performing access control checks on the return value of an advised method invocation; consider the case of a service that returns a list of document URI's... depending on the identity of the (Windows) user that is running the program that is calling this service, one could strip out those URI's that contain sensitive data for which the user does not have sufficient privileges to access. That is just one (real world) scenario... I'm sure you can think of plenty more that are a whole lot more relevant to your own development needs.

Here is the Spring.NET XML configuration for applying the 'after advice' declaratively...

34.3.1.3. Throws advice

So far we've covered 'around advice', 'before advice', and 'after advice'... these advice types will see you through most if not all of your AOP needs. However, one of the remaining advice types that Spring.NET has in its locker is 'throws advice'.

'throws advice' is advice that executes when an advised method invocation throws an exception.. hence the name. One basically applies the 'throws advice' to a target object in much the same way as any of the previously mentioned advice types. If during the execution of ones application none of any of the advised methods throws an exception, then the 'throws advice' will never execute. However, if during the execution of your application an advised method does throw an exception, then the 'throws advice' will kick in and be executed. You can use 'throws advice' to apply a common exception handling policy across the various objects in your application, or to perform logging of every exception thown by an advised method, or to alert (perhaps via email) the support team in the case of particularly of critical exceptions... the list of possible uses cases is of course endless.

The 'throws advice' type in Spring.NET is defined by the IThrowsAdvice interface in the Spring.Aop namespace... basically, one defines on one's 'throws advice' implementation class what types of exception are going to be handled. Lets take a quick look at the IThrowsAdvice interface...

```
public interface IThrowsAdvice : IAdvice
```

```
{
}
```

Yes, that is really it... it is a marker interface that has no methods on it. You may be wondering how Spring.NET determines which methods to call to effect the running of one's *'throws advice'*. An example would perhaps be illustrative at this point, so here is some simple Spring.NET style *'throws advice'*...

```
public class ConsoleLoggingThrowsAdvice : IThrowsAdvice
{
    public void AfterThrowing(Exception ex)
    {
        Console.Out.WriteLine("Advised method threw this }
}
```

Lets also change the implementation of the Execute() method of the ServiceCommand class such that it throws an exception. This will allow the advice encapsulated by the above ConsoleLoggingThrowsAdvice to kick in.

```
public class ServiceCommand : ICommand
{
    public void Execute()
    {
       throw new UnauthorizedAccessException();
    }
}
```

Let's programmatically apply the 'throws advice' (an instance of our ConsoleLoggingThrowsAdvice) to the invocation of the Execute() method of the above ServiceCommand class; to wit...

```
ProxyFactory factory = new ProxyFactory(new ServiceComma
factory.AddAdvice(new ConsoleLoggingThrowsAdvice());
ICommand command = (ICommand) factory.GetProxy();
command.Execute();
```

The result of executing the above snippet of code will look something like this...

Advised method threw this exception : System.Unauthorize Attempted to perform an unauthorized operation.

As can be seen from the output, the <code>ConsoleLoggingThrowsAdvice</code> kicked in when the advised method invocation threw an exception. There are a number of things to note about the <code>ConsoleLoggingThrowsAdvice</code> advice class, so lets take them each in turn.

In Spring.NET, 'throws advice' means that you have to define a class that implements the **IThrowsAdvice** interface. Then, for each type of exception that your 'throws advice' is going to handle, you have to define a method with this signature...

void AfterThrowing(Exception ex)

Basically, your exception handling method has to be named AfterThrowing. This name is important... your exception handling method(s) absolutely must be called AfterThrowing. If your handler method is not called AfterThrowing, then your 'throws advice' will never be called, it's as simple as that. Currently, this naming restriction is not configurable (although it may well be opened up for configuration in the future).

Your exception handling method must (at the very least) declare a parameter that is an Exception type... this parameter can be the root Exception class (as in the case of the above example), or it can be an Exception subclass if you only want to handle certain types of exception. It is good practice to always make your exception handling methods have an Exception parameter that is the most specialized Exception type possible... i.e. if you are applying 'throws advice' method could to that only a **ArgumentExceptions**, then declare the parameter of your exception handling method as...

void AfterThrowing(ArgumentException ex)

Note that your exception handling method can have any return type, but returning any value from a Spring.NET *'throws advice'* method would be a waste of time... the Spring.NET AOP infrastructure will simply ignore the return value,

so always define the return type of your exception handling methods to be **void**.

Finally, here is the Spring.NET XML configuration for applying the 'throws advice' declaratively...

One thing that cannot be done using 'throws advice' is exception swallowing. It is not possible to define an exception handling method in a 'throws advice' implementation that will swallow any exception and prevent said exception from bubbling up the call stack. The nearest thing that one can do is define an exception handling method in a 'throws advice' implementation that will wrap the handled exception in another exception; one would then throw the wrapped exception in the body of one's exception handling method. One can use this to implement some sort of exception translation or exception scrubbing policy, in which implementation specific exceptions (such as Sqlexception or OracleException exceptions being thrown by an advised data access object) get replaced with a business exception that has meaning to the service objects in one's business layer. A toy example of this type of 'throws advice' can be seen below.

```
// business objects in higher level service laye
throw new PersistenceException ("Cannot access p
}
}
```

Spring.NET's data access library already has this kind of functionality (and is a whole lot more sophisticated)... the above example is merely being used for illustrative purposes.

This treatment of 'throws advice', and of Spring.NET's implementation of it is rather simplistic. 'throws advice' features that have been omitted include the fact that one can define exception handling methods that permit access to the original object, method, and method arguments of the advised method invocation that threw the original exception. This is a quickstart guide though, and is not meant to be exhaustive... do consult the 'throws advice' section of the reference documentation, which describes how to declare an exception handling method that gives one access to the above extra objects, and how to declare multiple exception handling methods on the same IThrowsAdvice implementation class (see Section 13.3.2.3, "Throws advice").

34.3.1.4. Introductions (mixins)

In a nutshell, introductions are all about adding new state and behaviour to arbitrary objects... transparently and at runtime. Introductions (also called mixins) allow one to emulate multiple inheritance, typically with an eye towards applying crosscutting state and operations to a wide swathe of objects in your application that don't share the same inheritance hierarchy.

34.3.1.5. Layering advice

The examples shown so far have all demonstrated the application of a single advice instance to an advised object. Spring.NET's flavor of AOP would be pretty poor if one could only apply a single advice instance per advised object... it is perfectly valid to apply multiple advice to an advised object. For example, one might apply transactional advice to a service object, and also apply a security access checking advice to that same advised service object.

In the interests of keeping this section lean and tight, let's simply apply *all* of the advice types that have been previously described to a single advised object... in this first instance we'll just use the default pointcut which means that every

possible joinpoint will be advised, and you'll be able to see that the various advice instances are applied in order.

Please do consult the class definitions for the following previously defined advice types to see exactly what each advice type implementation does... we're going to be using single instances of the ConsoleLoggingAroundAdvice,

ConsoleLoggingBeforeAdvice,

ConsoleLoggingAfterAdvice,

and

ConsoleLoggingThrowsAdvice advice to advise a single instance of the ServiceCommand class.

You can find the following listing and executable application in the AopQuickStart solution in the project Spring.AopQuickStart.Step1.

```
ProxyFactory factory = new ProxyFactory(new ServiceComma
factory.AddAdvice(new ConsoleLoggingBeforeAdvice());
factory.AddAdvice(new ConsoleLoggingAfterAdvice());
factory.AddAdvice(new ConsoleLoggingThrowsAdvice());
factory.AddAdvice(new ConsoleLoggingAroundAdvice());
ICommand command = (ICommand) factory.GetProxy();
command.Execute();
```

Here is the Spring.NET XML configuration for declaratively applying multiple advice.

You can find the following listing and executable application in the AopQuickStart solution in the project Spring.AopQuickStart.Step2.

34.3.1.6. Configuring advice

In case it is not immediately apparent, remember that advice is just a plain old .NET object (a PONO); advice can have constructors that can take any number of parameters, and like any other .NET class, advice can have properties. What this means is that one can leverage the power of the Spring.NET IoC container to apply the IoC principle to one's advice, and in so doing reap all the benefits of Dependency Injection.

Consider the case of throws advice that needs to report (fatal) exceptions to a first line support centre. The throws advice could declare a dependency on a reporting service via a .NET property, and the Spring.NET container could dependency inject the reporting service dependency into the throws advice when it is being created; the reporting dependency might be a simple Log4NET wrapper, or a Windows EventLog wrapper, or a custom reporting exception reporting service that sends detailed emails concerning the fatal exception.

Also bear in mind the fact that Spring.NET's AOP implementation is quite independent of Spring.NET's IoC container. As you have seen, the various examples used in this have illustrated both programmatic and declarative AOP configuration (the latter being illustrated via Spring.NET's IoC XML configuration mechanism).

34.3.2. Using Attributes to define Pointcuts

34.4. The Spring.NET AOP Cookbook

The preceding treatment of Spring.NET AOP has (quite intentionally) been decidedly simple. The overarching aim was to convey the concepts of Spring.NET AOP... this section of the Spring.NET AOP guide contains a number of real world examples of the application of Spring.NET AOP.

34.4.1. Caching

This example illustrates one of the more common usages of AOP... caching.

Lets consider the scenario where we have some static reference data that needs to be kept around for the duration of an application. The data will almost never change over the uptime of an application, and it exists only in the database to satisfy referential integrity amongst the various relations in the database schema. An example of such static (and typically immutable) reference data would be a collection of <code>Country</code> objects (comprising a country name and a code). What we would like to do is suck in the collection of <code>Country</code> objects and then pin them in a cache. This saves us having to hit the back end database again and again every time we need to reference a country in our application (for example, to populate dropdown controls in a Windows Forms desktop application).

The Data Access Object (DAO) that will load the collection of <code>Country</code> objects is called <code>AdoCountryDao</code> (it is an implementation of the data-access-technology agnostic DAO interface called <code>ICountryDao</code>). The implementation of the <code>AdoCountryDao</code> is quite simple, in that every time the <code>FindAllCountries</code> instance method is called, an instance will query the database for an <code>IDataReader</code> and hydrate zero or more <code>Country</code> objects using the returned data.

```
public class AdoCountryDao : ICountryDao
{
    public IList FindAllCountries ()
    {
        // implementation elided for clarity...
        return countries;
    }
}
```

Ideally, what we would like to have happen is for the results of the *first* call to the FindAllCountries instance method to be cached. We would also like to do this in a non-invasive way, because caching is something that we might want to apply at any number of points across the codebase of our application. So, to address what we have identified as a *cross cutting concern*, we can use Spring.NET AOP to implement the caching.

The mechanism that this example is going to use to identify (or pick out) areas in our application that we would like to apply caching to is a .NET Attribute. Spring.NET ships with a number of useful custom .NET Attribute implementations, one of which is the cunningly named CacheAttribute. In the specific case of this example, we are simply going to decorate the definition of the FindAllCountries instance method with the CacheAttribute.

```
public class AdoCountryDao : ICountryDao
{
    [Cache]
    public IList FindAllCountries ()
    {
        // implementation elided for clarity...
        return countries;
    }
}
```

The SpringAir reference application that is packaged as part of the Spring.NET distribution comes with a working example of caching applied using Spring.NET AOP (see Chapter 37, SpringAir - Reference Application).

34.4.2. Performance Monitoring

This recipe show how easy it is to instrument the classes and objects in an application for performance monitoring. The performance monitoring implementation uses one of the (many) Windows performance counters to display and track the performance data.

34.4.3. Retry Rules

This final recipe describes a simple (but really quite useful) aspect... retry logic. Using Spring.NET AOP, it is quite easy to surround an operation such as a

method that opens a connection to a database with a (configurable) aspect that tries to obtain a database connection any number of times in the event of a failure.

34.5. Spring.NET AOP Best Practices

Spring.NET AOP is an 80% AOP solution, in that it only tries to solve the 80% of those cases where AOP is a good fit in a typical enterprise application. This final section of the Spring.NET AOP guide describes where Spring.NET AOP is typically useful (the 80%), as well as where Spring.NET AOP is not a good fit (the 20%).

Chapter 35. Portable Service Abstraction Quick Start

35.1. Introduction

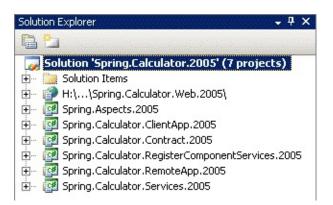
This quickstart demonstrates the basic usage of Spring.NET's portable service abstraction functionality. Sections 2-5 demonstrate the use of .NET Remoting, Section 6 shows the use of the ServicedComponentExporter for .NET Enterprise Services, and Section 7 shows the use of the WebServiceExporter.

35.2. .NET Remoting Example

The infrastructure classes are located in the Spring.Services assembly under the Spring.Services.Remoting namespace. The overall strategy is to export .NET objects on the server side as either CAO or SAO objects using CaoExporter or SaoExporter and obtain references to these objects on the client side using CaoFactoryObject and SaoFactoryObject. This quickstart does assume familiarity with .NET Remoting on the part of the reader. If you are new to .NET remoting you may find the links to introductory remoting material presented at the conclusion of this quickstart of some help.

As usual with quick start examples in Spring.NET, the classes used in the quickstart are intentionally simple. In the specific case of this remoting quickstart we are going to make a simple calculator that can be accessed remotely. The same calculator class will be exported in multiple ways reflecting the variety of .NET remoting options available (CAO, SAO-SingleCall, SAO-Singleton) and also the use of adding AOP advice to SAO hosted objects.

The example solution is located in the examples\Spring\Spring.Calculator directory and contains multiple projects.



The Spring.Calculator.Contract project contains the interface ICalculator that defines the basic operations of a calculator and another interface IAdvancedCalculator that adds support for memory storage for results. (woo hoo - big feature - HP-12C beware!) These interfaces are shown

below. The Spring.Calculator.Services project contains an implementation of the these interfaces, namely the classes Calculator and AdvancedCalculator. The purpose of AdvancedCalculator implementation is demonstrate to the configuration of object state for SAO-singleton objects. Note that the calculator implementations do not inherit from the MarshalByRefObject class. The Spring.Calculator.ClientApp project contains the client application and the Spring.Calculator.RemoteApp contains a console application that will host a Remoted instance of the AdvancedCalculator class. The Spring.Aspects project contains some logging advice that will be used to demonstrate the application of remoted aspects objects. Spring.Calculator.RegisterComponentServices related to enterprise service exporters and is not relevant for this quickstart. Spring.Calculator.Web is related to web services exporters and is not relevant for this quickstart.

```
public int Rest
{
         get { return _rest; }
         set { _rest = value; }
}
```

An extension of this interface that supports having a slot for calculator memory is shown below

```
public interface IAdvancedCalculator : ICalculator
{
    int GetMemory();
    void SetMemory(int memoryValue);
    void MemoryClear();
    void MemoryAdd(int num);
}
```

The structure of the VS.NET solution is a consequence of following the best practice of using interfaces to share type information between a .NET remoting client and server. The benefits of this approach are that the client does not need a reference to the assembly that contains the implementation class. Having the client reference the implementation assembly is undesirable for a variety of reasons. One reason being security since an untrusted client could potentially obtain the source code to the implementation since Intermediate Language (IL) code is easily reverse engineered. Another, more compelling, reason is to provide a greater decoupling between the client and server so the server can update its implementation of the interface in a manner that is quite transparent to the client; i.e. the client code need not change. Independent of .NET remoting best practices, using an interface to provide a service contract is just good objectoriented design. This lets the client choose another implementation unrelated to .NET Remoting, for example a local, test-stub or a web services implementation. One of the major benefits of using Spring.NET is that it reduces the cost of doing 'interface based programming' to almost nothing. As such, this best practice approach to .NET remoting fits naturally into the general approach to

application development that Spring.NET encourages you to follow. Ok, with that barrage of OO design ranting finished, on to the implementation!

35.3. Implementation

The implementation of the calculators contained in the Spring.Calculator.Servies project is quite straightforward. The only interesting methods are those that deal with the memory storage, which is the state that we will be configuring explicitly using constructor injection. A subset of the implementation is shown below.

```
public class Calculator : ICalculator
{
    public int Add(int n1, int n2)
    {
        return n1 + n2;
    }
    public int Substract(int n1, int n2)
        return n1 - n2;
    }
    public DivisionResult Divide(int n1, int n2)
        DivisionResult result = new DivisionResult();
        result.Quotient = n1 / n2;
        result.Rest = n1 % n2;
        return result;
    }
    public int Multiply(int n1, int n2)
        return n1 * n2;
    }
}
public class AdvancedCalculator : Calculator, IAdvancedCalcu
{
  private int memoryStore = 0;
```

```
public AdvancedCalculator()
{}

public AdvancedCalculator(int initialMemory)
{
    memoryStore = initialMemory;
}

public int GetMemory()
{
    return memoryStore;
}

// other methods omitted in this listing...
}
```

The Spring.Calculator.RemotedApp project hosts remoted objects inside a console application. The code is also quite simple and shown below

```
public static void Main(string[] args)
{
    try
    {
        // initialization of Spring.NET's IoC container
        IApplicationContext ctx = ContextRegistry.GetContext

        Console.Out.WriteLine("Server listening...");
    }
    catch (Exception e)
    {
        Console.Out.WriteLine(e);
    }
    finally
    {
        Console.Out.WriteLine("--- Press < return> to quit --
        Console.ReadLine();
    }
}
```

The configuration of the .NET remoting channels is done using the standard <code>system.runtime.remoting</code> configuration section inside the .NET configuration file of the application (<code>App.config</code>). In this case we are using the <code>tcp</code> channel on port 8005.

The objects created in Spring's application context are shown below. Multiple resource files are used to export these objects under various remoting configurations. The AOP advice used in this example is a simple Log4Net based around advice.

```
<configSections>
                 <sectionGroup name="spring">
                          <section name="context" type="Spring</pre>
                           <section name="objects" type="Spring</pre>
                           <section name="parsers" type="Spring</pre>
                  </sectionGroup>
                 <section name="log4net" type="log4net.Config</pre>
         </configSections>
<spring>
                  <parsers>
                          <parser type="Spring.Remoting.Confid")</pre>
                  </parsers>
         <context>
                 <resource uri="config://spring/objects" />
                 <resource uri="assembly://RemoteServer/Remot</pre>
                 <resource uri="assembly://RemoteServer/Remot</pre>
                 <resource uri="assembly://RemoteServer/Remot</pre>
                 <resource uri="assembly://RemoteServer/Remot</pre>
                 <resource uri="assembly://RemoteServer/Remot</pre>
         </context>
         <objects xmlns="http://www.springframework.net">
```

```
<description>Definitions of objects to be ex
    <object type="Spring.Remoting.RemotingConfigurer, Spring</pre>
      property name="Filename" value="Spring.Calculator.Ren
    </object>
                 <object id="Log4NetLoggingAroundAdvice" type</pre>
                          property name="Level" value="Debug"
                 </object>
                 <object id="singletonCalculator" type="Sprir</pre>
                          <constructor-arg type="int" value="2</pre>
                 </object>
                 <object id="singletonCalculatorWeaved" type=</pre>
                          property name="target" ref="singlet
                          property name="interceptorNames">
                                  st>
                                           <value>Log4NetLoggir
                                  </list>
                          </property>
                 </object>
                 <object id="prototypeCalculator" type="Spring"</pre>
                          <constructor-arg type="int" value="2</pre>
                 </object>
                 <object id="prototypeCalculatorWeaved" type=</pre>
                          property name="targetSource">
                                  <object type="Spring.Aop.Tar</pre>
                                           cproperty name="Targ
                                  </object>
                          </property>
                          property name="interceptorNames">
                                  st>
                                           <value>Log4NetLoggir
                                  </list>
                          </property>
                 </object>
        </objects>
</spring>
```

The declaration of the calculator instance, <code>singletonCalculator</code> for example, and the setting of any property values and / or object references is done as you would normally do for any object declared in the Spring.NET configuration file. To expose the calculator objects as .NET remoted objects the exporter <code>Spring.Remoting.CaoExporter</code> is used for CAO objects and <code>Spring.Remoting.SaoExporter</code> is used for SAO objects. Both exporters require the setting of a <code>TargetName</code> property that refers to the name of the object in Spring's IoC container that will be remoted. The semantics of <code>SAO-SingleCall</code> and CAO behavior are achieved by exporting a target object that is declared as a "prototype" (i.e. singleton=false). For <code>SAO</code> objects, the <code>ServiceName</code> property defines the name of the service as it will appear in the URL that clients use to locate the remote object. To set the remoting lifetime of the objects to be infinite, the property <code>Infinite</code> is set to true.

The configuration for the exporting a SAO-Singleton is shown below.

The configuration shown above uses the Spring Remoting schema but you can also choose to use the standard 'generic' XML configuration shown below.

```
<object name="saoSingletonCalculator" type="Spring.Remoting.
  <pre>cproperty name="TargetName" value="singletonCalculator" />
  cproperty name="ServiceName" value="RemotedSaoSingletonCalculator" />
</object>
```

This will result in the remote object being identified by the URL tcp://localhost:8005/RemotedSaoSingletonCalcula
The use of SaoExporter and CaoExporter for other configuration are

similar, look at the configuration files in the Spring.Calculator.RemotedApp project files for more information.

On the client side, the client application will connect a specific type of remote calculator service, object, ask it for it's current memory value, which is preconfigured to 217, then perform a simple addition. As in the case of the server, the channel configuration is done using the standard .NET Remoting configuration section of the .NET application configuration file (App.config), as can been seen below.

The client implementation code is shown below.

```
public static void Main(string[] args)
{
        try
        {
                Pause();
                IApplicationContext ctx = ContextRegistry.Ge
                Console.Out.WriteLine("Get Calculator...");
                IAdvancedCalculator firstCalc = (IAdvancedCa
    Console.WriteLine("Divide(11, 2) : " + firstCalc.Divide(
                Console.Out.WriteLine("Memory = " + firstCal
                firstCalc.MemoryAdd(2);
                Console.Out.WriteLine("Memory + 2 = " + firs
                Console.Out.WriteLine("Get Calculator...");
                IAdvancedCalculator secondCalc = (IAdvancedC
                Console.Out.WriteLine("Memory = " + secondCa
        catch (Exception e)
```

Note that the client application code is not aware that it is using a remote object. The Pause() method simply waits until the Return key is pressed on the console so that the client doesn't make a request to the server before the server has had a chance to start. The standard configuration and initialization of the .NET remoting infrastructure is done before the creation of the Spring.NET IoC container. The configuration of the client application is constructed in such a that easily switch implementations of way can calculatorService retrieved from the application context. In more complex applications the calculator service would be a dependency on another object in your application, say in a workflow processing layer. The following listing shows a configuration for use of a local implementation and then several remote implementations. The same Exporter approach can be used to create Web Services and Serviced Components (Enterprise Services) of the calculator object but are not discussed in this QuickStart.

```
<!-- <resource uri="assembly://Spring.Calculator.ClientA
   <!-- Web Service implementations -->
   <!-- Make sure 'http://localhost/SpringCalculator/' web
   <!-- <resource uri="assembly://Spring.Calculator.ClientA
   <!-- <resource uri="assembly://Spring.Calculator.ClientA
   <!-- EnterpriseService implementations -->
   <!-- Make sure you register components with 'RegisterCon
   <!-- <resource uri="assembly://Spring.Calculator.ClientA
 </context>
</spring>
```

The inProcess.xml configuration file creates an instance of AdvancedCalculator directly

```
<objects xmlns="http://www.springframework.net">
    <description>inProcess</description>
    <object id="calculatorService" type="Spring.Calculator.Ser"
</objects>
```

Factory classes are used to create a client side reference to the .NET remoting implementations. For SAO objects use the SaoFactoryObject class and for CAO objects use the CaoFactoryObject class. The configuration for obtaining a reference to the previously exported SAO singleton implementation is shown below

```
<objects xmlns="http://www.springframework.net">
     <description>saoSingleton</description>
```

You must specify the property ServiceInterface as well as the location of the remote object via the ServiceUrl property. The property replacement facilities of Spring.NET can be leveraged here to make it easy to configure the URL value based on environment variable settings, a standard .NET configuration section, or an external property file. This is useful to easily switch between test, QA, and production (yea baby!) environments. An example of how this would be expressed is...

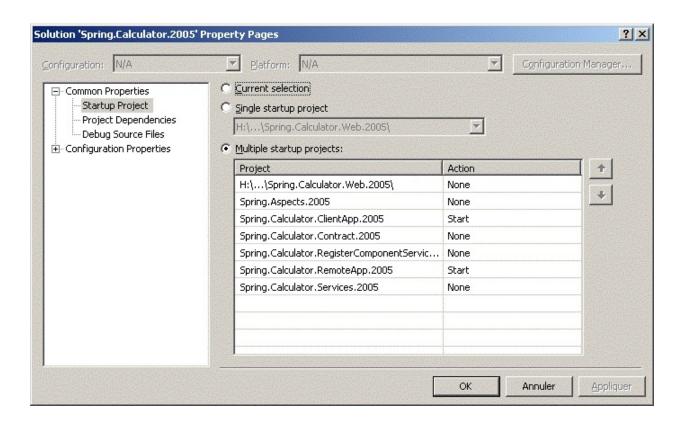
```
cproperty name="ServiceUrl" value="${protocol}://${host}:${protocol}://$
```

The property values in this example are defined elsewhere; refer to <u>Section 5.9.2.1</u>, <u>"Example: The PropertyPlaceholderConfigurer"</u> for additional information. As mentioned previously, more important in terms of configuration flexibility is the fact that now you can swap out different implementations (.NET remoting based or otherwise) of this interface by making a simple change to the configuration file.

The configuration for obtaining a reference to the previously exported CAO implementation is shown below

35.4. Running the application

Now that we have had a walk though of the implementation and configuration it is finally time to run the application (if you haven't yet pulled the trigger). Be sure to set up VS.NET to run multiple applications on startup as shown below.



Running the solution yields the following output in the server and client window

```
SERVER WINDOW

Server listening...
--- Press <return> to quit ---

CLIENT WINDOW

--- Press <return> to continue --- (hit return...)

Get Calculator...

Divide(11, 2): Quotient: '5'; Rest: '1'

Memory = 0
```

```
Memory + 2 = 2
Get Calculator...
Memory = 2
--- Press <return> to continue ---
```

35.5. Remoting Schema

The spring-remoting.xsd file in the doc directory provides a short syntax to configure Spring.NET remoting features. To install the schema in the VS.NET environment run the install-schema NAnt script in the doc directory. Refer to the Chapter on VS.NET integration for more details.

The various configuration files in the RemoteServer and Client projects show the schema in action. Here is a condensed listing of those definitions which should give you a good feel for how to use the schema.

```
<!-- Calculator definitions -->
<object id="singletonCalculator" type="Spring.Calculator.Ser</pre>
        <constructor-arg type="int" value="217" />
</object>
<object id="prototypeCalculator" type="Spring.Calculator.Ser</pre>
        <constructor-arg type="int" value="217" />
</object>
<!-- CAO object -->
<r:caoExporter targetName="prototypeCalculator" infinite="fa</pre>
  <r:lifeTime initialLeaseTime="2m" renewOnCallTime="1m"/>
</r:caoExporter>
<!-- SAO Single Call -->
<r:saoExporter
  targetName="prototypeCalculator"
  serviceName="RemotedSaoSingleCallCalculator"/>
<!-- SAO Singleton -->
<r:saoExporter
  targetName="singletonCalculator"
  serviceName="RemotedSaoSingletonCalculator" />
```

Note that the singleton nature of the remoted object is based on the Spring object definition. The "PrototypeCalculator" has its singleton property set to false to that a new one will be created every time a method on the remoted object is invoked for the SAO case.

35.6. .NET Enterprise Services Example

Enterprise Services example is located in the .NET project Spring.Calculator.RegisterComponentServices.2005.csproj or Spring.Calculator.RegisterComponentServices.2003.csproj, depending on the use of .NET 1.1 or 2.0. The example uses the previous AdvancedCalculator and then imports the embedded configuration implementation 'enterpriseServices.xml' the from namespace Spring.Calculator.RegisterComponentServices.Config. The level top configuration is shown below

The exporter that adapts the AdvancedCalculator for use as an Enterprise Service component is defined first in enterpriseServices.xml. Second is defined an exporter that will host the exported Enterprise Services component application by signing the assembly, registering it with the specified COM+ application name. If application does not exist it will create it and configure it using values specified for Description, AccessControl and Roles properties. The configuration file for enterpriseServices.xml is shown below

```
<object id="calculatorComponent" type="Spring.Enterp"</pre>
        roperty name="TargetName" value="calculate"
        cproperty name="TypeAttributes">
                st>
                        <object type="System.Enterpr</pre>
                </list>
        </property>
        property name="MemberAttributes">
                <dictionary>
                         <entry key="*">
                                 st>
                                         <object type
                                 </list>
                        </entry>
                </dictionary>
        </property>
</object>
<object type="Spring.EnterpriseServices.EnterpriseSe"</pre>
        property name="ApplicationName">
                <value>Spring Calculator Application
        </property>
        cproperty name="Description">
                <value>Spring Calculator applicatior
        </property>
        property name="AccessControl">
                <object type="System.EnterpriseServi</pre>
                        property name="AccessChecks
                                 <value>ApplicationCd
                        </property>
                </object>
        </property>
        property name="Roles">
                st>
                         <value>Admin : Administrator
                         <value>User : User role</val</pre>
                         <value>Manager : Administrat
                </list>
        </property>
        components">
                st>
                        <ref object="calculatorCompd"</pre>
```

35.7. Web Services Example

The WebServices example shows how to export the AdvancedCalculator as a web service that is an AOP proxy of AdvancedCalculator that has logging advice applied to it. The main configuration file, Web.config, includes information from three locations as shown below

The config section 'spring/objects' in Web.config contains the definition for the 'plain' Advanced calculator, as well as the definitions to create an AOP proxy of an AdvancedCalculator that adds logging advice. These definitions are shown below

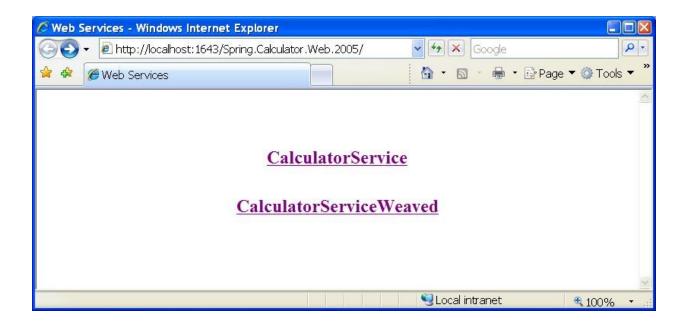
```
<objects xmlns="http://www.springframework.r</pre>
<!-- Aspect -->
<object id="CommonLoggingAroundAdvice" type="Spring.As</pre>
  cproperty name="Level" value="Debug"/>
</object>
<!-- Service -->
<!-- 'plain object' for AdvancedCalculator -->
<object id="calculator" type="Spring.Calculator.Service"</pre>
<!-- AdvancedCalculator object with AOP logging advice
<object id="calculatorWeaved" type="Spring.Aop.Framework")</pre>
                                cproperty name="target"
                                property name="intercer
                                          st>
                                                    <value
                                          </list>
                                </property>
                       </object>
```

```
</objects>
```

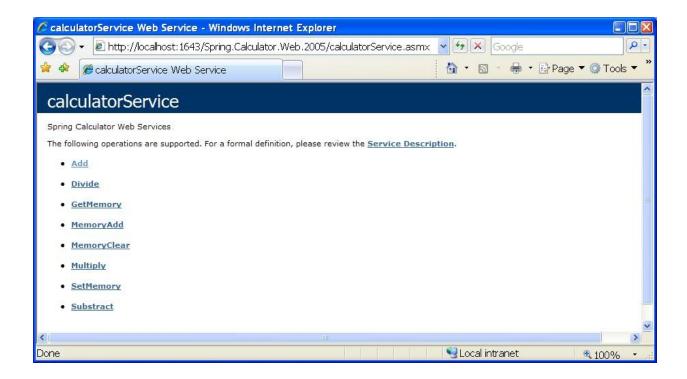
The configuration file webService.xml simply exports the named calculator object

Whereas the webService-aop.xml exports the calculator instance that has AOP advice applied to it.

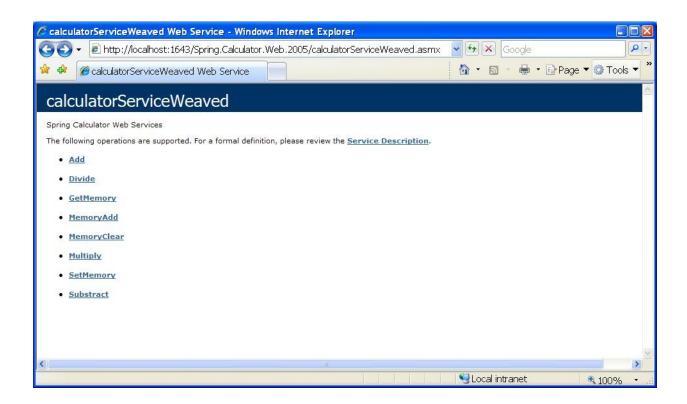
Setting the solution to run the web project as the startup, you will be presented with a screen as shown below



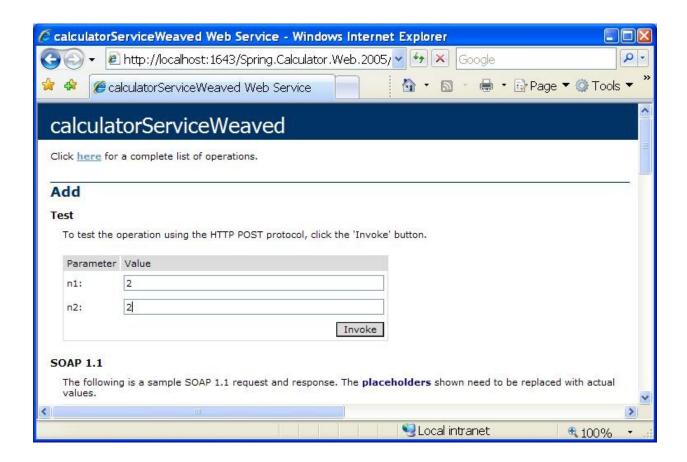
Selecting the CalculatorService and CalculatorServiceWeaved links will bring you to the standard user interface generated for browsing a web service, as shown below



And similarly for the calculator service with AOP applied



Invoking the Add method for calculatorServiceWeaved shows the screen



Invoking add will then show the result '4' in a new browser instance and the log file log.txt will contain the following entires

```
2007-10-15 17:59:47,375 [DEBUG] Spring.Aspects.Logging.Commc 2007-10-15 17:59:47,421 [DEBUG] Spring.Aspects.Logging.Commc
```

35.8. Additional Resources

Some introductory articles on .NET remoting can be found online at MSDN. Ingo Rammer is also a very good authority on .NET remoting, and the .NET Remoting FAQ (link below) which is maintained by Ingo is chock full of useful information.

- An Introduction to Microsoft .NET Remoting Framework
- Microsoft .NET Remoting: A Technical Overview
- Advanced .NET Remoting (authored by Ingo Rammer)
- .NET Remoting FAQ

Chapter 36. Web Quickstarts

36.1. Introduction

The Web Quickstart solution provides basic 'Hello World' examples for using Spring. Web features. You can use this solution as a starting point and then move on to the SpringAir application that uses a wider range of Spring. Web features.

Chapter 37. SpringAir - Reference Application

37.1. Introduction

The SpringAir sample application demonstrates a selection of Spring.NET's powerful features making a .NET programmer's life easier. It demonstrates the following features of Spring.Web

- Spring.NET IoC container configuration
- Dependency Injection as applied to ASP.NET pages
- Master Page support
- Web Service support
- Bi-directional data binding
- Declarative validation of domain objects
- Internationalization
- Result mapping to better encapsulate page navigation flows

The application models a flight reservation system where you can browse flights, book a trip and even attach your own clients by leveraging the web services exposed by the SpringAir application.

All pages within the application are fully Spring managed. Dependencies get injected as configured within a Spring Application Context. For NET 1.1 it shows how to apply centrally managed layouts to all pages in an application by using master pages - a well-known feature from NET 2.0.

When selecting your flights, you are already experiencing a fully localized form. Select your preferred language from the bottom of the form and see, how the new language is immediately applied. As soon as you submit your desired flight, the submitted values are automatically unbound from the form onto the application's data model by leveraging Spring.Web's support for Data Binding. With Data Binding you can easily associate properties on your PONO model with elements on your ASP.NET form.

37.2. Getting Started

The application is located in the installation directory under 'examples/SpringAir. The directory 'SpringAir.Web.2003' contains the .NET 1.1 version of the application and the directory 'SpringAir.Web.2005' contains the .NET 2.0 version. For .NET 1.1 you will need to create a virtual directory named 'SpringAir.2003' using IIS Administrator and point it to the following directory examples\Spring\SpringAir\src\SpringAir.Web.2003. The solution file for .NET 1.1 is examples\Spring\SpringAir\SpringAir.2003.sln. For .NET 2.0 simply open the solution examples\Spring\SpringAir\SpringAir.2005.sln. Set your startup project to be SpringAir.Web and the startpage to .\Web\Home.aspx

37.3. Container configuration

The web project's top level Web.config configures the IoC container that is used within the web application. You do not need to explicitly instantiate the IoC container. The important parts of that configuration are shown below

```
<spring>
    <parsers>
        <parser type="Spring.Data.Config.DatabaseNamespacePa</pre>
    </parsers>
    <context>
        <resource uri="~/Config/Aspects.xml"/>
        <resource uri="~/Config/Web.xml"/>
        <resource uri="~/Config/Services.xml"/>
        <!-- TEST CONFIGURATION -->
       <resource uri="~/Config/Test/Services.xml"/>
       <resource uri="~/Config/Test/Dao.xml"/>
       <!-- PRODUCTION CONFIGURATION -->
       <resource uri="~/Config/Production/Services.xml"/>
       <resource uri="~/Config/Production/Dao.xml"/>
       -->
    </context>
</spring>
```

In this example there are separate configuration files for test and production configuration. The Services.xml file is in fact the same between the two, and the example will be refactored in future to remove that duplication. The Dao layer in the test configuration is an in-memory version, faking database access, whereas the production version uses an ADO.NET based solution.

The pages that comprise the application are located in the directory 'Web/BookTrip'. In that directory is another Web.config that is responsible for configuring that directory's .aspx pages. There are three main pages in the flow of the application.

- TripForm form to enter in airports, times, round-trip or one-way
- Suggested Flights form to select flights
- ReservationConfirmationPage your confirmation ID from the booking process.

The XML configuration to configure the TripForm form is shown below

As you can see the various services it needs are set using standard DI techniques. The Results property externalizes the page flow, redirecting to the next page in the flow, SuggestedFlights. The 'parent' attribute lets this page inherit properties from a template. The is located in the top level Web.config file, packaged under the Config directory. The standardPage sets up properties of Spring's base page class, from which all the pages in this application inherit from. (Note that to perform only dependency injection on pages you do not need to inherit from Spring's Page class).

37.4. Bi-directional data binding

The TripForm page demonstrates the bi-directional data binding features. A Trip object is used to back the information of the form. The family of methods that are overridden to support the bi-directional data binding are listed below.

```
protected override void InitializeModel()
 {
                 trip = new Trip();
                 trip.Mode = TripMode.RoundTrip;
                 trip.StartingFrom.Date = DateTime.Today;
                 trip.ReturningFrom.Date = DateTime.Today.AddDays(1);
}
protected override void LoadModel(object savedModel)
 {
                 trip = (Trip)savedModel;
}
protected override object SaveModel()
                 return trip;
}
protected override void InitializeDataBindings()
 {
                 BindingManager.AddBinding("tripMode.Value", "Trip.Mode.Value", "Trip.M
                 BindingManager.AddBinding("leavingFromAirportCode.Se
                 BindingManager.AddBinding("goingToAirportCode.Select
                 BindingManager.AddBinding("leavingFromDate.Selected[
                 BindingManager.AddBinding("returningOnDate.Selected[
}
```

This is all you need to set up in order to have values from the Trip object 'marshaled' to and from the web controls. The InitializeDataBindings method set this up, using the Spring Expression Language to define the UI element property that is associate with the model (Trip) property.

37.5. Declarative Validation

The method called when the Search button is clicked will perform validation. If validation succeeds as well as additional business logic checks, the next page in the flow is loaded. This is shown in the code below. Notice how much cleaner and more business focused the code reads than if you were using standard ASP.NET APIs.

```
protected void SearchForFlights(object sender, EventArgs
{
    if (Validate(trip, tripValidator))
    {
        FlightSuggestions suggestions = this.bookingAger
        if (suggestions.HasOutboundFlights)
        {
            Session[Constants.SuggestedFlightsKey] = sug
            SetResult(DisplaySuggestedFlights);
        }
    }
}
```

The 'Validate' method of the page takes as arguments the object to validate and a IValidator instance. The TripForm property TripValidator is set via dependency injection (as shown above). The validation logic is defined declaratively in the XML configuration file and is shown below.

```
<v:group id="departureDateVa
                 <v:required test="St
            <v:message id="error.der
          </v:required>
                 <v:condition test="5</pre>
            <v:message id="error.der
                 </v:condition>
        </v:group>
        <v:group id="returnDateValid
                 <v:required test="Re
            <v:message id="error.ret
          </v:required>
                 <v:condition test="F
            <v:message id="error.ret
          </v:condition>
        </v:group>
</v:group>
```

The validation logic has 'when' clauses so that return dates can be ignored if the Mode property of the Trip object is set to 'RoundTrip'.

37.6. Internationalization

Both image and text based internationalization are supported. You can see this in action by clicking on the English, Srpski, or Српски links on the bottom of the page.

37.7. Web Services

The class BookingAgent that was used by the TripForm class is a standard .NET class, i.e no WebMethod attributes are on any of its methods. Spring can expose this object as a web service by declaring the following XML defined in the top level Config/Services.xml file

```
<object id="bookingAgentWebService" type="Spring.Web.Servi</pre>
  cproperty name="TargetName" value="bookingAgent"/>
  property name="Name" value="BookingAgent"/>
  property name="Namespace" value="http://SpringAir/WebSe
  property name="Description" value="SpringAir Booking Ad
  property name="MemberAttributes">
    <dictionary>
     <entry key="SuggestFlights">
       <object type="System.Web.Services.WebMethodAttribu"</pre>
         property name="Description" value="Gets those f
       </object>
      </entry>
      <entry key="Book">
       <object type="System.Web.Services.WebMethodAttribu"</pre>
         roperty name="Description" value="Goes ahead a
       </object>
      </entry>
      <entry key="GetAirportList">
       <object type="System.Web.Services.WebMethodAttribu"</pre>
         col
       </object>
     </entry>
    </dictionary>
  </property>
</object>
```

Chapter 38. Data Access QuickStart

38.1. Introduction

The data access quick start demonstrates the API usage of AdoTemplate (both generic and non-generic versions) as well as the use of the object based data access classes contained in Spring.Data.Objects. It uses the Northwind database and is located under the directory examples/DataAccessQuickStart.

The quick start contains pseudo DAO objects and a collection of NUnit tests to exercise them rather than a full blown application. To run the tests from within VS.NET install <u>TestDriven.NET</u>, <u>ReSharper</u>, or an equivalent. The listing of DAO classes and the parts of Spring.Data that they demonstrate is shown below.

- CommandCallbackDao Use of the ICommandCallback and CommandCallbackDelegate
- ResultSetExtractorDao Use of IResultSetExtractor and ResultSetExtractorDelegate
- RowCallbackDao Use of IRowCallback and RowCallbackDelegate
- RowMapperDao Use of IRowMapper and RowMapperDelegate
- QueryForObject Use of QueryForObject method.
- StoredProcDao Use of Spring.Data.Objects.StoredProcedure

The are simple domain objects in the Spring.DataQuickStart.Domain namespace, collections of which are generally returned from the DAO methods.

38.1.1. Database configuration

To get started running the 'unit test' you should configure the database connection string. The listing in DataQuickStart.GenericTemplate.ExampleTests.xml is shown below

```
</objects>
```

You should change the value of the provider element to correspond to you database and the connection string as appropriate. Please refer to the documentation on the <u>DbProvider</u> abstraction for details particular to your database configuration. You should also install the Northwind database, which is available for SqlServer 2005 from this <u>download location</u>. The minimal schema to support other database providers may be supported in the future.

38.1.1.1. AdoTemplate Configuration

The various DAO objects refer to an instance of AdoTemplate which is responsible for performing data access operations. This is declared in ExampleTest.xml as shown below

The property DbProvider refers to the database configuration you previously defined. Also the property DataReaderWrapper is set to the NullMappingDataReader that ships with Spring. This provides convenient default values for null values returned from the database. To read more about AdoTemplate, refer to the chapter, Data access using ADO.NET.

38.1.2. CommandCallback

The code that exercises the use of a CommandCallback is shown below

```
[Test]
public void CallbackDaoTest()
{
    CommandCallbackDao commandCallbackDao = ctx["com
    int count = commandCallbackDao.FindCountWithPost
    Assert.AreEqual(3, count);
}
```

The configuration of the CommandCallbackDao is shown below

```
<object id="commandCallbackDao" type="Spring.DataQuickStar
  <pre>cproperty name="AdoTemplate" ref="adoTemplate"/>
  </object>
```

This the minimal configuration required for a DAO object, typically DAO objects in your application will include other configuration information, for example properties to specify the maximum size of the result set returned etc. The implementation of the FindCountWithPostalCode is shown below

Anonymous delegates are used to specify the implementation of the callback function that passes in a DbCommand object. You can then use the DbCommand object as you see fit to access the database. If you are using Spring's delcarative transaction management features then this DbCommand would have its transaction and connection properties based on the context of the surrounding transaction. All resource management for the DbCommand are handled for you

by the framework, as well as error reporting on error etc. If you execute the test, it will pass, assuming you haven't modified any data in the Northwind database from its raw installation.

Chapter 39. Transactions QuickStart

39.1. Introduction

The Transaction Quickstart demonstrates Spring's transaction management features. The database schema are two simple tables, credit and debit, which contain an Identifier and an Amount. The quick start shows the use of declarative transactions using attributes and also the ability to change the transaction manager (local or distributed) via changes to only the configuration files - no code changes are required. It also demonstrates some techniques for unit and integration testing an application as well as separating Spring's configuration files so that one is responsible for describing how the core business classes are configured and others that are responsible for the database environment and application of AOP.

This quickstart assumes you have installed a way to run NUnit tests within your IDE. Some excellent tools that let you do this are <u>TestDriven.NET</u> and <u>ReSharper</u>.

39.2. Application Overview

The design of the application is very simple and consists of two logical layers, a business service layer in the namespace Spring.TxQuickStart.Services and a DAO layer in the namespace Spring.TxQuickStart.Dao. As this is just a toy example the business service layer does nothing more than call two DAO objects. The business service is to transfer money in a bank account and is blatantly taken from the book **Pro ADO.NET** by Sahil Malik. The transfer service is defined by the interface IACCountManager with the implementation AccountManager the located in namespace Spring.TxQuickStart.Services. The money is recorded in a credit and debit table in the database. The SQL Server schema for the tables is located in the file CreditsDebitsSchema.sql. Transferring the money requires an ACID operation on these two tables. The credit operation is defined via a IACCOUNTCreditDao interface and the debit operation via IACCOUNTDebitDao interface. Implementations of these interfaces using AdoTemplate are in the namespace Spring.TxQuickStart.Dao.Ado.

39.2.1. Interfaces

The Manager and DAO interfaces are shown below

```
public interface IAccountManager
{
    void DoTransfer(float creditAmount, float debitAmour)}

public interface IAccountCreditDao
{
    void CreateCredit(float creditAmount);
}

public interface IAccountDebitDao
{
    void DebitAccount(float debitAmount);
}
```

39.3. Implementation

The implementation of the Account Credit DAO is shown below

and for the Debit DAO

Both of these DAO implementations inherit from Spring's AdoDaoSupport class that provides convenient access to an AdoTemplate for performing data access operations. With no other properties that can be configured in these implementations, the only configuration required is setting of AdoDaoSupport's DbProvider property representing the connection to the database.

The implementation of the service layer interface, **IACCOUNTManager**, is shown below.

```
public class AccountManager : IAccountManager
{
    private IAccountCreditDao accountCreditDao;
    private IAccountDebitDao accountDebitDao;
```

```
private float maxTransferAmount = 1000000;
    public AccountManager(IAccountCreditDao accountCredi
        this.accountCreditDao = accountCreditDao;
        this.accountDebitDao = accountDebitDao;
    }
    public float MaxTransferAmount
        get { return maxTransferAmount; }
        set { maxTransferAmount = value; }
    }
    [Transaction]
    public void DoTransfer(float creditAmount, float det
        accountCreditDao.CreateCredit(creditAmount);
        if (creditAmount > maxTransferAmount || debitAmount
        {
            throw new ArithmeticException("see a teller
        }
        accountDebitDao.DebitAccount(debitAmount);
    }
}
```

The if statement is a poor-mans representation of business logic, namely that there is a policy that does not allow the use of this service for amounts larger than \$1,000,000. If the credit or debit amount is larger than 1,000,000 then and exception will be thrown. We can write a unit test that will test for this business logic and provide stub implementations of the DAO objects so that our tests are not only independent of the database but will also execute very quickly.



Note

Notice the Transaction attribute on the DoTransfer method. This attribute can be read by Spring and used to create a transactional

proxy to AccountManager in order to perform declarative transaction management.

The NUnit unit test for AccountManager is shown below

```
public class AccountManagerUnitTests
{
    private IAccountManager accountManager;
    [SetUp]
    public void Setup()
        IAccountCreditDao stubCreditDao = new StubAccour
        IAccountDebitDao stubDebitDao = new StubAccountD
        accountManager = new AccountManager(stubCreditDa
    }
    [Test]
    public void TransferBelowMaxAmount()
        accountManager.DoTransfer(217, 217);
    }
    [Test]
    [ExpectedException(typeof(ArithmeticException))]
    public void TransferAboveMaxAmount()
        accountManager.DoTransfer(2000000, 200000);
}
```

Running these tests we exercise both code pathways through the method <code>DOTransfer</code>. Nothing we have done so far is Spring specific (aside from the presence of the [Transaction] attribute. Now that we know the class works in isolation, we can now 'wire' up the application for use in production by specifying how the service and DAO layers are related. This configuration file is shown below and can loosely be referred to as your 'application blueprint'. This configuration file is named application-config.xml and is an embedded resource inside the 'main' project, Spring.TxQuickStart.

```
<objects xmlns='http://www.springframework.net'>
```

This configuration is selecting the real ADO.NET implementations that will insert records into the database. We can now write a NUnit integration test that will test the service and DAO layers. To do this we add on configuration information specific to our test environment. This extra configuration information will determine what databases we speak to and what transaction manager (local or distribute) to use. The code for this integration style NUnit test is shown below

```
[TestFixture]
public class AccountManagerTests
{
    private AdoTemplate adoTemplateCredit;
    private AdoTemplate adoTemplateDebit;

    private IAccountManager accountManager;

    [SetUp]
    public void SetUp()
    {
        // Configure Spring programmatically
        NamespaceParserRegistry.RegisterParser(typeof(Dage))
```

```
NamespaceParserRegistry.RegisterParser(typeof(Tx
        NamespaceParserRegistry.RegisterParser(typeof(Ad
        IApplicationContext context = new XmlApplication
            "assembly://Spring.TxQuickStart.Tests/Spring
        accountManager = context["accountManager"] as IA
        CleanDb(context);
    }
    [Test]
    public void TransferBelowMaxAmount()
        accountManager.DoTransfer(217, 217);
        int numCreditRecords = (int)adoTemplateCredit.Ex
        int numDebitRecords = (int)adoTemplateDebit.Exe
        Assert.AreEqual(1, numCreditRecords);
        Assert.AreEqual(1, numDebitRecords);
    }
    [Test]
    [ExpectedException(typeof(ArithmeticException))]
    public void TransferAboveMaxAmount()
    {
        accountManager.DoTransfer(2000000, 200000);
    }
    private void CleanDb(IApplicationContext context)
        IDbProvider dbProvider = (IDbProvider)context["E
        adoTemplateDebit = new AdoTemplate(dbProvider);
        adoTemplateDebit.ExecuteNonQuery(CommandType.Tex
        dbProvider = (IDbProvider)context["CreditDbProvi
        adoTemplateCredit = new AdoTemplate(dbProvider);
        adoTemplateCredit.ExecuteNonQuery(CommandType.Te
    }
}
```

The essential element is to create an instance of Spring's application context

where the relevant layers of the application are 'wired' together. The **IACCOUNTManager** implementation is retrieved from the IoC container and stored as a field of the test class. The basic logic of the test is the same as in the unit test but in addition there is the verification of actions performed in the database. The set up method puts the database tables into a known state before running the tests. Other techniques for performing integration testing that can alleviate the need to do extensive database state management for integration tests is described in the testing section.

39.4. Configuration

The configuration file system-test-local-config.xml shown in the previous program listing includes application-config.xml and specifies the database to use and the local (not distributed) transaction manager AdoPlatformTransactionManager. This configuration file is shown below

```
<objects xmlns="http://www.springframework.net"</pre>
         xmlns:db="http://www.springframework.net/database"
         xmlns:tx="http://www.springframework.net/tx">
  <!-- Imports application configuration -->
  <import resource="assembly://Spring.TxQuickStart/Spring.Tx</pre>
  <!-- Imports additional aspects -->
  <import resource="assembly://Spring.TxQuickStart.Tests/Spr</pre>
  <!-- Database Providers -->
  <db:provider id="DebitDbProvider"</pre>
               provider="System.Data.SqlClient"
               connectionString="Data Source=MARKT60\SQL2005
  <db:provider id="CreditDbProvider"</pre>
               provider="System.Data.SqlClient"
               connectionString="Data Source=MARKT60\SQL2005
  <alias name="DebitDbProvider" alias="CreditDbProvider"/>
  <!-- Transaction Manager if using a single database that d
  <object id="transactionManager"</pre>
          type="Spring.Data.Core.AdoPlatformTransactionManad
    cproperty name="DbProvider" ref="DebitDbProvider"/>
  </object>
  <!-- Transaction aspect -->
```

```
<tx:attribute-driven/>
</objects>
```

Moving from top to bottom in the configuration file, the 'application-blueprint' configuration file is included. Then the database type and connection parameters are specified for the two databases. The names of these providers must match those specific in application-config.xml. Since the two names point to the same database, an alias configuration element is used to have them point to the same dbProvider under different names. The type of transaction manager is then selected, in this case we are showing the use of local transactions with AdoPlatformTransactionManager. Running the tests will result in 217 being entered into the Credits and Debits table of each database. You can fire up SQL Server Management Studio or equivalent to verify this.

To switch to a distributed transaction you can refer to the configuration file system-test-dtc-config.xml, which is shown below

TxScopeTransactionManager uses .NET 2.0 System.Transactions as the implementation, allowing for distributed transactions between the two different databases listed. In a larger application the different layers would typically be broken up into individual configuration files and imported into the main configuration file. This allows your configuration to mirror your architecture.

You can also use the configuration file system-test-dtc-es-config.xml that will use EnterpriseServices to perform transaction management.

39.4.1. Rollback Rules

Using Rollback rules allows you to specify which exceptions will not cause a rollback and instead only stop execution flow, committing the work done up to the exception. An alternative implementation of AccountManager's DoTransfer method (included in the sample code) is shown below.

```
[Transaction(NoRollbackFor = new Type[] { typeof(Ari
public void DoTransfer(float creditAmount, float deb
{
    accountCreditDao.CreateCredit(creditAmount);

    if (creditAmount > maxTransferAmount || debitAmount);
        throw new ArithmeticException("see a teller)
    }

    accountDebitDao.DebitAccount(debitAmount);
}
```

All that has changed is the use of the NoRollbackFor property on the transaction attribute.

The expected behavior is that the credit table will be updated even though the exception is thrown. This is due to specifying that exceptions of the type ArithmethicException should not rollback the database transaction. Running the test code below verifies that the exception still propagates out of the method.

```
[Test]
public void DeclarativeWithAttributesNoRollbackFor()
{
    try
    {
        accountManager.DoTransfer(2000000, 2000000);
        Assert.Fail("Should have thrown Arithmetic E
    } catch (ArithmeticException) {
        int numCreditRecords = (int)adoTemplateCredith int numDebitRecords = (int)adoTemplateDebit.
        Assert.AreEqual(1, numCreditRecords);
        Assert.AreEqual(0, numDebitRecords);
    }
}
```

39.5. Adding additional Aspects

Transactional advice is just one type of advice that can be applied to the service layer. You can also configure other pieces of advice to be executed as part of the general advice chain that is associated with methods that have the Transaction attribute applied. In this example we will add logging of thrown exceptions using Spring's ExceptionHandlerAdvice as well as logging of the service layer method invocation. No code is required to be changed in order to have this additional functionality. Instead all you have to do is uncomment the line

```
<import resource="assembly://Spring.TxQuickStart.Tests/Spr</pre>
```

in either system-test-dtc-config.xml or system-test-local-config.xml The aspect configuration file is shown below

```
<objects xmlns='http://www.springframework.net'</pre>
         xmlns:aop="http://www.springframework.net/aop">
  <object name="exceptionAdvice" type="Spring.Aspects.Except</pre>
    property name="exceptionHandlers">
      st>
        <value>on exception name ArithmeticException log 'Log'
      </list>
    </property>
  </object>
  <object name="loggingAdvice" type="Spring.Aspects.Logging."</pre>
    cproperty name="logUniqueIdentifier" value="true"/>
    property name="logExecutionTime"
                                          value="true"/>
    property name="logMethodArguments"
                                          value="true"/>
                                          value=";"/>
    property name="Separator"
    property name="HideProxyTypeNames"
                                          value="true"/>
    property name="UseDynamicLogger"
                                          value="true"/>
    property name="LogLevel"
                                          value="Info"/>
  </object>
```

The transaction aspect is now additionally configured with an order value of "10", which will place it after the execution of the exception aspect, which is configured to use an order value of 1. The behavior for logging the exception is configuring of specified by creating and an instance Spring.Aspects.Exceptions.ExceptionHandlerAdvic The location where that behavior is applied, the pointcut, is the Transaction attribute. The logging of method arguments and execution time is specified by configuring of instance Spring.Aspects.Logging.SimpleLoggingAdvice.

The AOP configuration section on the bottom is what ties together the behavior and where it will take place in the program flow. Under the covers the transaction configuration, <tx:attribute-driven/> creates similar advice and pointcut definitions. Running the test TransferBelowMaxAmount will then log the following messages

```
INFO - Entering DoTransfer;45b6af04-b736-4efa-a489-45462726
INFO - Exiting DoTransfer;45b6af04-b736-4efa-a489-454627260
```

When the test case of the test TransferAboveMaxAmount is run the following messages are logged

```
INFO - Entering DoTransfer;d94bc81b-a4ff-4ca1-9aaa-f2834f26
INFO - Exception thrown in DoTransferDoTransfer;d94bc81b-a4
System.ArithmeticException: see a teller big spender...
    at Spring.TxQuickStart.Services.AccountManager.DoTransfer
    at Spring.DynamicReflection.Method_DoTransfer_ec48557f22b
    at Spring.Reflection.Dynamic.SafeMethod.Invoke(Object tar
    at Spring.Aop.Framework.DynamicMethodInvocation.InvokeJoi
    at Spring.Aop.Framework.AbstractMethodInvocation.Proceed(
    at Spring.Transaction.Interceptor.TransactionInterceptor.
    at Spring.Aop.Framework.AbstractMethodInvocation.Proceed(
    at Spring.Aop.Framework.AbstractMethodInvocation.Proceed(
    at Spring.Aspects.Logging.SimpleLoggingAdvice.InvokeUnder
TRACE - Logging an exception thrown from method DoTransfer
```

Chapter 40. NHibernate QuickStart

40.1. Introduction

This QuickStart application uses the all too familiar Northwind database and uses NHibernate browse and edit customers. It It is a very simple application that directly uses the DAO layer in some cases and also has a simple service layer that simulated a fullillment process. See the integration tests as well for insight into how it works. The application uses Spring's declarative transaction management features, HibernateTemplate helper class, and Open Session In View module.



The example will be updated to not use HibernateTemplate and instead use the standard NHibernate API in a future release. All functionality is still present when using the standard NHibernate API, as Spring transaction managment is integrated into NHibernate extension points and exception translation is provided by AOP advice. See the section titled Implementing DAOs based on plain Hibernate 1.2/2.0 API" in the hibernate orm section of the reference docs for more information.

To run the application make the Web application the project that starts and set Default.aspx as the start page. You will see a list of customers. If you select 'edit' then you can edit some customer info and save it by pressing the save button. Note that you will need to explicitly navigate back to the Default.aspx page and reload it in order to see the changes.

Chapter 41. Quartz QuickStart

41.1. Introduction

In many applications the need arises to perform a certain action at a given time without any user interaction, usually to perform some administrative tasks. These tasks need to be scheduled, say to perform a job in the early hours of the morning before the start of business. This functionality is provided by using job scheduling software. Quartz.NET is an excellent open source job scheduler that can be used for these purposes. It provides a wealth of features, such as persistent jobs and clustering. To find out more about Quartz.NET visit their web site. Spring integration allows you to use Spring to configure Quartz jobs, triggers, and schedulers and also provides integration with Spring's transaction management features.

The full details of Quartz are outside the scope of this quickstart but here is 'quick tour for the impatient' of the main classes and interfaces used in Quartz so you can get your sea legs. A Quartz IJob interface represents the task you would like to execute. You either directly implement Quartz's IJob interface or a convenience base class. The Quartz Trigger controls when a job is executed, for example in the wee hours of the morning every weekday. This would be done using Quartz's CronTrigger implementation. Instances of your job are created every time the trigger fires. As such, in order to pass information between different job instances you stash data away in a hashtable that gets passed to the each Job instance upon its creation. Quartz's JobDetail class combines the IJOb and this hashtable of data. Instead of System.Collections.Hashtable the standard JobDataMap is used. Triggers are registered with a Quartz IScheduler implementation that manages the overall execution of the triggers and jobs. The StdSchedulerFactory implementation is generally used.

41.2. Application Overview

The sample application has two types of Jobs. One that inherits from Spring's convenience base class <code>QuartzJobObject</code> and another which does not inherit from any base class. The latter class is adapted by Spring to be a Job. Two triggers, one for each of the jobs, are created. These triggers are in turn registered with a scheduler. In each case the job implementation will write information to the console when it is executed.

41.3. Standard job scheduling

The Spring base class <code>QuartzJobObject</code> implements <code>IJob</code> and allows for your object's properties to be set via values that are stored inside Quartz's <code>JobDataMap</code> that is passed along each time your job is instantiated due a trigger firing. This class is shown below

```
public class ExampleJob : QuartzJobObject
{
    private string userName;
    public string UserName
    {
        set { userName = value; }
    }
    protected override void ExecuteInternal(JobExecution {
            Console.WriteLine("{0}: ExecuteInternal called, DateTime.Now, userName, context.NextFireTime }
}
```

The method <code>ExecuteInternal</code> is called when the trigger fires and is where you would put your business logic. The <code>JobExecutionContext</code> passed in lets you access various pieces of information about the current job execution, such as the <code>JobDataMap</code> or information on when the next time the trigger will fire. The <code>ExampleJob</code> is configured by creating a <code>JobDetail</code> object as shown below in the following XML snippet taken from spring-objects.xml

```
</dictionary>
</property>
</object>
```

The dictionary property of the JobDetailObject, JobDataAsMap, is used to set the values of the ExampleJob's properties. This will result in the ExampleJob being instantiated with it's UserName property value set to 'Alexandre' the first time the trigger fires.

We then will schedule this job to be executed on 20 second increments of every minute as shown below using Spring's CronTriggerObject which creates a Quartz CronTrigger.

Lastly, we schedule this trigger with the scheduler as shown below

Running this configuration will produce the following output

```
8/8/2008 1:29:40 PM: ExecuteInternal called, user name: Alex 8/8/2008 1:30:00 PM: ExecuteInternal called, user name: Alex 8/8/2008 1:30:20 PM: ExecuteInternal called, user name: Alex
```

41.4. Scheduling arbitrary methods as jobs

It is very convenient to schedule the execution of method as a job. The AdminService class in the example demonstrates this functionality and is listed below.

```
public class AdminService
{
    private string userName;

    public string UserName
    {
        set { userName = value; }
    }

    public void DoAdminWork()
    {
        Console.WriteLine("{0}: DoAdminWork called, user)
}
```

Note that it does not inherit from any base class. To instruct Spring to create a <code>JobDetail</code> object for this method we use Spring's factory object class <code>MethodInvokingJobDetailFactoryObject</code> as shown below

Note that AdminService object is configured using Spring as you would do normally, without consideration for Quartz. The trigger associated with the jobDetail object is listed below. Also note that when using MethodInvokingJobDetailFactoryObject you can't use database persistence for

Jobs. See the class documentation for additional details.

This creates an instances of Quartz's SimpleTrigger class (as compared to its CronTrigger class used in the previous section). StartDelay and RepeatInterval properties are TimeSpan objects than can be set using the convenient strings such as 10s, 1h, etc, as supported by Spring's custom TypeConverter for TimeSpans.

This trigger can then be added to the scheduler's list of registered triggers as shown below.

The interleaved output of both these jobs being triggered is shown below.

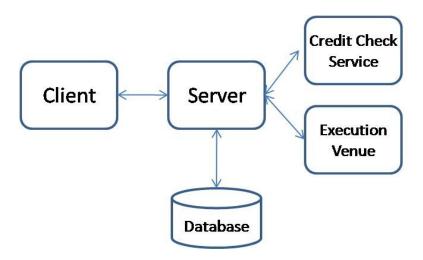
```
8/8/2008 1:40:18 PM: DoAdminWork called, user name: Gabriel 8/8/2008 1:40:20 PM: ExecuteInternal called, user name: Alex 8/8/2008 1:40:23 PM: DoAdminWork called, user name: Gabriel 8/8/2008 1:40:28 PM: DoAdminWork called, user name: Gabriel 8/8/2008 1:40:33 PM: DoAdminWork called, user name: Gabriel 8/8/2008 1:40:38 PM: DoAdminWork called, user name: Gabriel 8/8/2008 1:40:40 PM: ExecuteInternal called, user name: Alex 8/8/2008 1:40:43 PM: DoAdminWork called, user name: Gabriel 8/8/2008 1:40:43 PM: DoAdminWork called, user name: Gabriel 8/8/2008 1:40:53 PM: DoAdminWork called, user name: Gabriel 8/8/2008 1:40:53 PM: DoAdminWork called, user name: Gabriel
```

8/8/2008 1:40:58 PM: DoAdminWork called, user name: Gabriel 8/8/2008 1:41:00 PM: ExecuteInternal called, user name: Alex 8/8/2008 1:41:03 PM: DoAdminWork called, user name: Gabriel

Chapter 42. NMS QuickStart

42.1. Introduction

The NMS quick start application demonstrates how to use asynchronous messaging to implement a system for purchasing a stock. To purchase a stock, a client application will send a stock request message containing the information about the stock, i.e. ticker symbol, quantity, etc. The client request message will be received by the server where it will perform business processing on the request, for example to determine if the user has sufficient credit to purchase the stock or if the user is even allowed to make the purchase due to existing account restrictions. These are typically external processes as well. Usually the server application will persist state about the request and forward it on to an execute venue where the actual execution of the stock request is performed. In addition, market data for the stock will be sent from the server process to the client. The high level exchange of information is shown below.



This example was developed with ActiveMQ 5.1 and the ActiveMQ NMS library with subversion repository number 685750.

42.2. Message Destinations

To implement this flow using messaging the following queues and topics will be used. All requests from the client to the server will be sent on the queue named APP.STOCK.REQUEST. Responses to the requests will be sent from the server to the client on a queue unique to each client. In this example the queue name is of the form APP.STOCK.<UserName>, and more specifically is configured to be APP.STOCK.JOE. Market data does not need to be delivered to an individual client as many client applications are interested in this shared information. As such, the server will send market data information on a topic named APP.STOCK.MARKETDATA. The messaging communication between the server and the execution venue is not included as part of the application. An local implementation of the service interface that represents the execution venue is used instead of one based on messaging or another middleware technology. The messaging flow showing the queues and topics used is shown below.



Queues are shown in red and topics in green.

42.3. Gateways

Gateways represent the service operation to send a message. The client will send a stock request to the server based on the contract defined by the IStockService interface.

```
public interface IStockService
{
    void Send(TradeRequest tradeRequest);
}
```

The server will send market data to the clients based on the contract defined by the IMarketDataService interface.

```
public interface IMarketDataService
{
    void SendMarketData();
}
```

The market data gateway has no method parameters as it is assumed that implementations will manage the data to send internally. The <code>TradeRequest</code> object is one of the data objects that will be exchanged in the application and is discussed in the next section.

The use of interfaces allows for multiple implementations to be created. Implementations that use messaging to communicate will be based on the Spring's NmsGateway class and will be discussed later. stub or mock implementations can be used for testing purposes.

42.4. Message Data

The TradeRequest object shown above contains all the information required to process a stock order. To promote the interoperability of this data across different platforms the TradeRequest class is generated from an XML Schema using Microsoft's Schema Definition Tool (xsd.exe). The schema for trade request is shown below

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" eleme</pre>
           targetNamespace="http://www.springframework.net/r
  <xs:element name="TradeRequest">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Ticker" type="xs:string"/>
        <xs:element name="Quantity" type="xs:long"/>
        <xs:element name="Price" type="xs:decimal"/>
        <xs:element name="OrderType" type="xs:string"/>
        <xs:element name="AccountName" type="xs:string"/>
        <xs:element name="BuyRequest" type="xs:boolean"/>
        <xs:element name="UserName" type="xs:string"/>
        <xs:element name="RequestID" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Running xsd.exe on this schema will result in a class that contains properties for each of the element names. A partial code listing of the TradeRequest class is shown below

```
// This code was generated by a tool.
public partial class TradeRequest {
    public string Ticker {
        get {
            return this.tickerField;
        }
        set {
```

```
this.tickerField = value;
}

public long Quantity {
    get {
        return this.quantityField;
    }
    set {
        this.quantityField = value;
    }
}

// Additional properties not shown for brevity.
}
```

The schema and the TradeRequest class are located in the project Spring.NmsQuickStart.Common. This common project will be shared between the server and client for convenience.

When sending a response back to the client the type TradeResponse will be used. The schema for the TradeResponse is shown below

The **TradeResponse** type also generated from a schema using xsd.exe. A partial code listing is shown below

```
This code was generated by a tool.
//
    public partial class TradeResponse {
        public string Ticker {
            get {
                return this.tickerField;
            set {
                this.tickerField = value;
            }
        }
        public long Quantity {
            get {
                return this.quantityField;
            set {
                this.quantityField = value;
            }
        }
      // Additional properties not shown for brevity.
    }
```

The market data information will be sent using a Hashtable data structure.

42.5. Message Handlers

When the TradeRequest message is received by the server, it will be handled by the class Spring.NmsQuickStart.Server.Handlers.StockAppHa shown below

```
public class StockAppHandler
{
    private IExecutionVenueService executionVenueService
    private ICreditCheckService creditCheckService;
    private ITradingService tradingService;
    public TradeResponse Handle(TradeRequest tradeReques
    {
        TradeResponse tradeResponse;
        IList errors = new ArrayList();
        if (creditCheckService.CanExecute(tradeRequest,
            tradeResponse = executionVenueService.Execut
            tradingService.ProcessTrade(tradeRequest, tr
        else
            tradeResponse = new TradeResponse();
            tradeResponse.Error = true;
            tradeResponse ErrorMessage = errors[0].ToStr
        return tradeResponse;
    }
}
```

The stub implementations of the services, located in the namespace Spring.NmsQuickStart.Server.Services.Stubs, will result in always sending back a error-free trade response message. A realistic implementation would likely have the execution venue and trading service be remote services and the trading service could be implemented as a local transactional service layer that uses spring's declarative transaction management

features.

The client will receive a TradeResponse message as well as a Hashtable of data representing the market data. The message handle for the client is the class Spring.NmsQuickStart.Client.Handlers.StockAppHandler and is shown below.

```
public class StockAppHandler
{

    // definition of stockController omitted for brevity

    public void Handle(Hashtable data)
    {

            // forward to controller to update view
            stockController.UpdateMarketData(data);
     }

    public void Handle(TradeResponse tradeResponse)
    {

            // forward to controller to update view
            stockController.UpdateTrade(tradeResponse);
     }
}
```

What is important to note about these handlers is that they contain no messaging API artifacts. As such you can write unit and integration tests against these classes independent of the middleware. The missing link between the messaging world and the objects processed by the message handlers are message converters. Spring's messaging helper classes, i.e. SimpleMessageListenerContainer and NmsTemplate use message converters to pass data to the handlers and to send data via messaging for gateway implementations

42.6. Message Converters

The implementation of **IMessageConverter** is used Spring.NmsQuickStart.Common.Converters.XmlMessa This converter adds the ability to marshal and unmarshal objects to and from XML strings. It also uses Spring's SimpleMessageConverter to convert Hashtables, strings, and byte arrays. In order to pass information about the serialized type, type information is put in the message properties. The type information can be either the class name or an integer value identifying the type. In systems where the client and server are deployed together and are tightly coupled, sharing the class name is a convenient shortcut. The alternative is to register a type for a given integer value. The XML configuration used to configure these objects is shown below

This configuration is common between the server and the client.

42.7. Messaging Infrastructure

The implementations of the gateway interfaces inherit from Spring's helper class NmsGatewaySupport in order to get easy access to a NmsTemplate for sending. The implementation of the IStockService interface is shown below

```
public class NmsStockServiceGateway : NmsGatewaySupport,
    private IDestination defaultReplyToQueue;

    public IDestination DefaultReplyToQueue
    {
        set { defaultReplyToQueue = value; }
    }

    public void Send(TradeRequest tradeRequest)
    {
        NmsTemplate.ConvertAndSendWithDelegate(tradeRequest)
    }
}
```

The Send method is using NmsTemplate's ConvertAndSendWithDelegate(object obj, MessagePostProcessorDelegate messagePostProcessorDelegate) method. The anonymous delegate allows you to modify the message properties, such as NMSReplyTo and NMSCorrelationID after the message has been converted from an object but before it has been sent. The use of an anonymous delegate allows makes it very easy to apply any post processing logic to the converted message.

The object definition for the NmsStockServiceGateway is shown below along with its dependent object definitions of NmsTemplate and the ConnectionFactory.

```
<object name="StockServiceGateway" type="Spring.NmsQuickSt</pre>
  property name="NmsTemplate" ref="NmsTemplate"/>
  property name="DefaultReplyToQueue">
    <object type="Apache.NMS.ActiveMQ.Commands.ActiveMQQue"</pre>
      <constructor-arg value="APP.STOCK.JOE"/>
    </object>
  </property>
</object>
<object name="NmsTemplate" type="Spring.Messaging.Nms.Core</pre>
  property name="ConnectionFactory" ref="ConnectionFactor
  property name="DefaultDestinationName" value="APP.STOCK
  property name="MessageConverter" ref="XmlMessageConvert
</object>
<object id="ConnectionFactory" type="Apache.NMS.ActiveMQ.C</pre>
  <constructor-arg index="0" value="tcp://localhost:61616"</pre>
</object>
```

In this example the 'raw' Apache.NMS.ActiveMQ.ConnectionFactory connection factory was used. It would be more efficient resource wise to use Spring's CachingConnectionFactory wrapper class so that connections will not be open and closed for each message send as well as allowing for the caching of other intermediate NMS API objects such as sessions and message producers.

A similar configuration is used on the server to configure the class Spring.NmsQuickStart.Server.Gateways.MarketData that implements the IMarketDataService interface.

Since the client is also a consumer of messages, on the topic APP.STOCK.MARKETDATA and the queue APP.STOCK.JOE (for Trader Joe!), two message listener containers are defined as shown below.

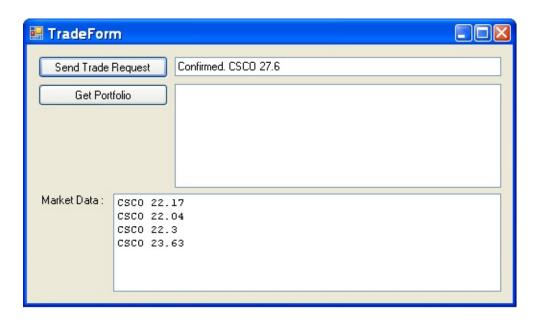
Refer to the messages reference docs for all the available attributes to configure the

container and also the section on registering the NMS schema with Spring..

On the server we define a message listener container for the queue APP.STOCK.REQUEST but set the concurrency property to 10 so that 10 threads will be consuming messages from the queue.

42.8. Running the application

To run both the client and server make sure that you select 'Multiple Startup Projects' within VS.NET. The GUI has a button to make a hardcoded trade request and show confirmation in a text box. A text area is used to display the market data. There is a 'Get Portfolio' button that is not implemented at the moment. A picture of the GUI after it has been running for a while and trade has been sent and responded to is shown below



Chapter 43. MSMQ QuickStart

43.1. Introduction

The MSMQ quick start application demonstrates how to use asynchronous messaging to implement a system for purchasing a stock. Is follows the same basic approach as in the NMS QuickStart but is adapted as need for use with MSMQ. Please read the introduction in that chapter to get an overview of the system.

When there is direct overlap in functionality between the MSMQ and NMS quickstart a reference to the appropriate section in the NMS QuickStart documentation is given.

43.2. Message Destinations

To communicate between th client and server a pair of queues will be used. Messages sent from the client to the server will use the transactional queue named .\Private\request.txqueue. Messages sent from the the client will use the transactional server to queue .\Private\$\response.joe.txqueue. The queue for messages that cannot be processed, so called 'poison messages' will be sent to the queue .\Private\$\dead.queue. You can create these queues using the computer management administration console. Private queues are used to simplify the application setup requirements.

Since MSMQ does not natively support the publish-subscribe messaging style as in other messaging systems, Apache MQ, IBM Websphere MQ, TIBCO EMS, the market data information is sent on the same queue as the responses from the server to the client for trade requests..

43.3. Gateways

The gateway interfaces are the same as those described in the NMS QuickStart here.

43.4. Message Data

TradeRequest and TradeResponse messages are defined using XML Schema and classes are generated from that schema. This is the same approach as described in more details in the NMS QuickStart here.

An important difference in the types of message data formats supported 'out-of-the-box' with Apache, IBM, TIBCO as compared to Microsoft MSMQ is the latter support sending a hashtable data structure. As a result, the hashtable that was used to send market data information from the server to the client was changed to be of type System. String in the MSMQ example.

43.5. Message Handlers

The message handlers are the same as used in the NMS QuickStart here, aside from the change of the hashtable data structure to a string. This is an important benefit of enforcing a separation between the messaging specific classes and the business processing layer.

43.6. MessageConverters

The message converter used is Spring.Messaging.Support.Converters.XmlMessageConverter. It is configured by specifying the data types that will be send and received. Here is a configuration example for types generated from the XML Schema and a plain string.

43.7. Messaging Infrastructure

The implementations of the gateway interfaces inherit from Spring's helper class MessageQueueGatewaySupport in order to get easy access to a MessageQueueTemplate for sending. The implementation of the IStockService interface is shown below

```
public class MsmqStockServiceGateway : MessageQueueGatewaySu
{
    private Random random = new Random();
    private string defaultResponseQueueObjectName;
    public string DefaultResponseQueueObjectName
        set { defaultResponseQueueObjectName = value; }
    }
    public void Send(TradeRequest tradeRequest)
        MessageQueueTemplate.ConvertAndSend(tradeReguest, de
    }
    private MessageQueue GetResponseQueue()
        return MessageQueueFactory.CreateMessageQueue(defaul
    }
}
```

The Send method is using MessageQueueTemplate's ConvertAndSend(object obj, MessagePostProcessorDelegate messagePostProcessorDelegate) method. The anonymous

delegate allows you to modify the message properties, such as ResponseQueue and AppSpecific after the message has been converted from an object but before it has been sent. The use of an anonymous delegate allows makes it very easy to apply any post processing logic to the converted message.

The configuration for MsmqStockServiceGateway and all its dependencies is shown below, highlighting important dependency links.

```
<object name="stockServiceGateway" type="Spring.MsmgQuickSta"</pre>
  property name="MessageQueueTemplate" ref="messageQueueTem"
  property name="DefaultResponseQueueObjectName" value="res
</object>
<object id="messageQueueTemplate" type="Spring.Messaging.Cor</pre>
  roperty name="DefaultMessageQueueObjectName" value="requ
  property name="MessageConverterObjectName" value="xmlMess
</object>
<object id="xmlMessageConverter" type="Spring.Messaging.Supp</pre>
  property name="TargetTypes">
    st>
      <value>Spring.MsmgQuickStart.Common.Data.TradeRequest,
      <value>Spring.MsmqQuickStart.Common.Data.TradeResponse
      <value>System.String, mscorlib</value>
    </list>
  </property>
</object>
<object id="requestTxQueue" type="Spring.Messaging.Support.N</pre>
  property name="Path" value=".\Private$\reguest.txgueue"/>
  property name="MessageReadPropertyFilterSetAll" value="tr
</object>
<object id="responseTxQueue" type="Spring.Messaging.Support.</pre>
  roperty name="Path" value=".\Private$\response.joe.txque
  property name="MessageReadPropertyFilterSetAll" value="tr
</object>
```

Since the client also needs to listen to incoming messages on the responseTxQueue,

TransactionalMessageListenerContainer is configured.

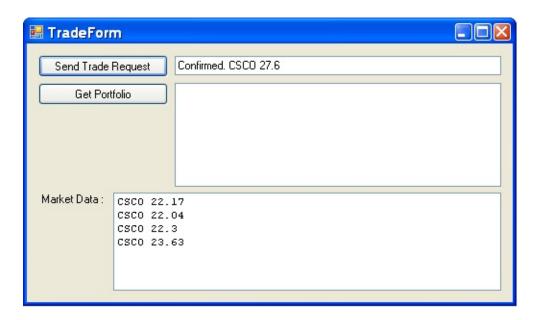
The configuration for the message listener container and all its dependencies is shown below, highlighting important dependency links.

```
<!-- MSMQ Transaction Manager -->
<object id="messageQueueTransactionManager" type="Spring.Mes</pre>
<!-- Message Listener Container that uses MSMQ transactional
<object id="transactionalMessageListenerContainer" type="Spr</pre>
  roperty name="MessageQueueObjectName" value="responseTxQ"
  property name="PlatformTransactionManager" ref="messageQu"
  property name="MessageListener" ref="messageListenerAdapt
  property name="MessageTransactionExceptionHandler" ref="s
</object>
<!-- Delegate to plain .NET object for message handling -->
<object id="messageListenerAdapter" type="Spring.Messaging.L</pre>
  property name="HandlerObject" ref="stockAppHandler"/>
  property name="DefaultHandlerMethod" value="Handle"/>
  property name="MessageConverterObjectName" value="xmlMess
</object>
<object id="sendToQueueExceptionHandler" type="Spring.Messag</pre>
  roperty name="MessageQueueObjectName" value="deadTxQueue
</object>
<object id="deadTxQueue" type="Spring.Messaging.Support.Mess</pre>
  cproperty name="Path" value=".\Private$\dead.gueue"/>
  property name="MessageReadPropertyFilterSetAll" value="tr
</object>
```

A similar configuration is used on the server to configure the class Spring.MsmqQuickStart.Server.Gateways.MarketData that implements the IMarketDataService interface and a TransactionalMessageListenerContainer to process messages on the requestTxQueue. You can increase the number of processing thread in the TransactionalMessageListenerContainer by setting the property MaxConcurrentListeners, the default value is 1.

43.8. Running the application

To run both the client and server make sure that you select 'Multiple Startup Projects' within VS.NET. The GUI has a button to make a hard coded trade request and show confirmation in a text box. A text area is used to display the market data. There is a 'Get Portfolio' button that is not implemented at the moment. A picture of the GUI after it has been running for a while and trade has been sent and responded to is shown below.



Chapter 44. WCF QuickStart

44.1. Introduction

The WCF quickstart application shows how to configure your WCF services using dependency injection and how to to apply AOP advice to your services. It is based on the same interfaces used in the <u>portable service abstractions quickstart</u> example that demonstrates similar features for .NET Remoting, Enterprise Services, and ASMX web sevices. The quickstart example is only available as a VS.NET 2008 solution.

There are two server applications in the solution, one is a web application where the WCF service will be hosted, and the other is a self-hosting console application, Spring.WcfQuickStart.Server.2008. The client application is located in Sprng.WcfQuickStart.ClientApp.2008. To run the solution make sure that all three projects are set to startup.

44.2. The server side

The service contract is shown below

```
[ServiceContract(Namespace = "http://Spring.WcfQuickStar
public interface ICalculator
{
     [OperationContract]
     double Add(double n1, double n2);
     [OperationContract]
     double Subtract(double n1, double n2);
     [OperationContract]
     double Multiply(double n1, double n2);
     [OperationContract]
     double Divide(double n1, double n2);
     [OperationContract]
     string GetName();
}
```

and the implementation is straightforward, only adding a property that controls how long each method should sleep. An abbreviated listing of the implementation is shown below

```
public class CalculatorService : ICalculator
{
    private int sleepInSeconds;

    public int SleepInSeconds
    {
        get { return sleepInSeconds; }
        set { sleepInSeconds = value; }
}

    public double Add(double n1, double n2)
    {
        Thread.Sleep(sleepInSeconds*1000);
        return n1 + n2;
}
```

44.2.1. WCF Dependency Injection and AOP in self-hosted application

The approach using dynamic proxies is used in the console application inside the Spring.WcfQuickStart.Server.2008 project. For more information on this approach refer to this <u>section</u> in the reference docs. The configuration of your service is done as you would typically do with Spring, including applying of any AOP advice. The class is hosted inside the console application through the use of Spring's <code>ServiceHostFactoryObject</code> exporter. The configuration for the server console application is shown below.

```
<objects xmlns="http://www.springframework.net"</pre>
         xmlns:aop="http://www.springframework.net/aop">
  <!-- Service definition -->
  <object id="calculator" singleton="false"</pre>
          type="Spring.WcfQuickStart.CalculatorService,
    property name="SleepInSeconds" value="1"/>
  </object>
 <object id="serviceOperation" type="Spring.Aop.Support</pre>
    roperty name="pattern" value="Spring.WcfQuickStart
 </object>
 <object id="perfAdvice" type="Spring.WcfQuickStart.Sim"</pre>
    roperty name="Prefix" value="Service Layer Perform
  </object>
  <aop:config>
    <aop:advisor pointcut-ref="serviceOperation" advice-</pre>
  </aop:config>
  <!-- host the service object -->
  <object id="calculatorServiceHost" type="Spring.ServiceHost"</pre>
    property name="TargetName" value="calculator" />
  </object>
```

</objects>

Look at the standard WCF configuration section in App.config for additional configuration details. In that section you will see that the name of the WCF service corresponds to the name of the service object inside the spring container.

44.2.2. WCF Dependency Injection and AOP in IIS web application

Much of the configuration ob the objects is the same as before, the .svc file though refers to the name of the service inside the Spring container as well as using Spring's Spring.ServiceModel.Activation.ServiceHostFactory. The .svc file is shown below.

44.3. Client access

The project Spring.WcfQuickStart.ClientApp.2008 is a console application that calls the two WCF services. It creates a client side proxy based on using ChannelFactory<T>.CreateChannel. Running the client application produces the following output.

```
--- Press <return> to continue ---
Web Calculator
Add(1, 1) : 2
Divide(11, 2) : 5.5
Multiply(2, 5) : 10
Subtract(7, 4) : 3

ServerApp Calculator
Add(1, 1) : 2
Divide(11, 2) : 5.5
Multiply(2, 5) : 10
Subtract(7, 4) : 3

--- Press <return> to continue ---
```

Spring Framework



Part VIII. Spring.NET for Java developers

This part of the reference documentation is for Java developers who would like a quick orientation to what is different between the Java and .NET versions of the framework.

• Chapter 45, Spring.NET for Java Developers

Chapter 45. Spring.NET for Java Developers

45.1. Introduction

This chapter is to help Java developers get their sea legs using Spring.NET. It is not intended to be a comprehensive comparison between .NET and Java. Rather, it highlights the day-to-day differences you will experience when you start to use Spring.NET.

45.2. Beans to Objects

There are some simple name changes, basically everywhere you saw the word 'bean' you will now see the word 'object'. A comparison of a simple Spring configuration file highlights these small name changes. Here is the application.xml file for the sample MovieFinder application in Spring.Java

Here is the corresponding file in Spring.NET

As you can easily see the <beans> and <bean> elements are replaced by <objects> and <object> elements. The class definition in Spring.Java contains the fully qualified class name. The Spring.NET version also contains the fully qualified classname but in addition specifies the name of the assembly where that type is located. This is necessary since .NET does not have a 'classpath' concept. Assembly names in .NET can have up to four parts to describe the exact version.

The other XML Schema elements in Spring.NET are the same as in Spring.Java's DTD except for specifying string based key value pairs. In Java this is represented by the java.util.Properties class and the xml element is name props> as shown below

In .NET the analogous class is System.Collections.Specialized.NameValueCollection and is represented by the xml element <name-values>. The listing of the elements also follows the .NET convention of application configuration files using the <add> element with 'key' and 'value' attributes. This is show below

45.3. PropertyEditors to TypeConverters

PropertyEditors from the java.beans package provide the ability to convert from a string to an instance of a Java class and vice-versa. For example, to set a string array property, a comma delimited string can be used. The Java class that provides this functionality is the appropriately named StringArrayPropertyEditor. In .NET, **TypeConverters** the from System.ComponentModel namespace provide the same functionality. The type conversion functionality in .NET also allows for TypeConverters to be explicitly registered with a data type. This allows for transparent setting of complex object properties. However, some classes in the .NET framework do not support the style of conversion we are used to from Spring. Java, such as setting of a string[] with a comma delimited string. The type converter, StringArrayConverter in the Spring.Objects.TypeConverters namespace is therefore explicitly registered with Spring.NET in order to provide this functionality. As in the case of Spring.Java, Spring.NET allows user defined type converters to be registered. However, if you are creating a custom type in .NET, using the standard .NET mechanisms for type conversion is the preferred approach.

45.4. ResourceBundle-ResourceManager

45.5. Exceptions

Exceptions in Java can either be checked or unchecked. .NET supports only unchecked exceptions. Spring.Java prefers the use of unchecked exceptions, frequently making conversions from checked to unchecked exceptions. In this respect Spring.Java is similar to the default behavior of .NET

45.6. Application Configuration

In Spring.Java it is very common to create an ObjectFactory or ApplicationContext from an external XML configuration file This functionality is also provided in Spring.NET. However, in .NET the System.Configuration namespace provides support for managing application configuration information. The functionality in this namespace depends on the availability of specially named files: Web.config for ASP.NET applications and <MyExe>.exe.config for WinForms and console applications. <MyExe> is the name of your executable. As part of the compilation process, if you have a file name App.config in the root of your project, the compiler will rename the file to <MyExe>.exe.config and place it into the runtime executable folder.

These application configuration files are XML based and contain configuration sections that can be referenced by name to retrieve custom configuration objects. In order to inform the .NET configuration system how to create a custom configuration object from one of these sections, an implementation of the interface, IConfigurationSectionHandler, needs to be registered. Spring.NET provides two implementations, one to create an IApplicationContext from a <context> section and another to configure the context with object definitions contained in an <objects> section. The <context> section is very powerful and expressive. It provides full support for locating all IResource via Uri syntax and hierarchical contexts without coding or using more verbose XML as would be required in the current version of Spring.Java

The <configSections> and <section> elements are a standard part of the .NET application configuration file. These elements are used to register an instance of IConfigurationSectionHandler and associate it with another xml element in the file, in this case the <context> and <objects> elements.

The following code segment is used to retrieve the IApplicationContext from the .NET application configuration file.

In order to enforce the usage of the named configuration section Spring/context the preferred instantiation mechanism is via the use of the registry class ContextRegistry as shown below

```
IApplicationContext ctx = ContextRegistry.GetContext();
```

45.7. AOP Framework

45.7.1. Cannot specify target name at the end of interceptorNames for ProxyFactoryObject

of When configuring the list interceptor names on a ProxyFactoryObject instance (or object definition), one cannot specify the name of the target (i.e. the object being proxied) at the end of the list of interceptor names. This shortcut is valid in Spring Java, where the ProxyFactoryBean will automatically detect this, and use the last name in the interceptor names list as the target of the ProxyFactoryBean. The following configuration, which would be valid in Spring Java (barring the obvious element name changes), is **not** valid in Spring.NET (so don't do it).

In Spring.NET, the InterceptorNames property of the ProxyFactoryObject can *only* be used to specify the names of interceptors. Use the TargetName property to specify the name of the target object that is to be proxied.

The main reason for not supporting exactly the same style of configuration as Spring Java is because this 'feature' is regarded as a legacy holdover from Rod Johnson's initial Spring AOP implementation, and is currently only kept as-is (in Spring Java) for reasons of backward compatibility.

Appendix A. XML Schema-based configuration

A.1. Introduction

This appendix details the use of XML Schema-based configuration in Spring.

The 'classic' **Object/>**-based schema is good, but its generic-nature comes with a price in terms of configuration overhead. Creating a custom XML Schema-based configuration makes Spring XML configuration files substantially clearer to read. In addition, it allows you to express the intent of an object definition.

The key thing to remember is that creating custom schema tags work best for infrastructure or integration objects: for example, AOP, collections, transactions, integration with 3rd-party frameworks, etc., while the existing object tags are best suited to application-specific objects, such as DAOs, service layer objects, etc.

Please note the fact that the XML configuration mechanism is totally customisable and extensible. This means you can write your own domain-specific configuration tags that would better represent your application's domain; the process involved in doing so is covered in the appendix entitled Appendix B. Extensible XML authoring.

A.2. XML Schema-based configuration

A.2.1. Referencing the schemas

As a reminder, you reference the standard objects schema as shown below

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"</pre>
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instand
         xsi:schemaLocation="http://www.springframework.net
<!-- <object/> definitions here -->
</objects>
```



The 'xsi:schemaLocation' fragment is not actually required, but can be included to reference a local copy of a schema (which can be useful during development) and assumes the XML editor will look to that location and load the schema.

The above Spring XML configuration fragment is boilerplate that you can copy and paste (!) and then plug <0bject/> definitions into like you have always done. However, the entire point of using custom schema tags is to make configuration easier.

The rest of this chapter gives an overview of custom XML Schema based configuration that are included with the release.



As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

A.2.2. The tx (transaction) schema

The **t**X tags deal with configuring objects in Spring's comprehensive support for transactions. These tags are covered in the chapter entitled Chapter 17, Transaction

management.



Tip

You are strongly encouraged to look at the 'spring-tx-1.1.xsd' file that ships with the Spring distribution. This file is (of course), the XML Schema for Spring's transaction configuration, and covers all of the various tags in the tx namespace, including attribute defaults and suchlike. This file is documented inline, and thus the information is not repeated here in the interests of adhering to the DRY (Don't Repeat Yourself) principle.

In the interest of completeness, to use the tags in the <code>tx</code> schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the <code>tx</code> namespace are available to you.



Note

Often when using the tags in the <code>tx</code> namespace you will also be using the tags from the <code>aop</code> namespace (since the declarative transaction support in Spring is implemented using AOP). The above XML snippet contains the relevant lines needed to reference the <code>aop</code> schema so that the tags in the <code>aop</code> namespace are available to you.

You will also need to configure the AOP and Transaction namespace parsers in the main .NET application configuration file as shown below



As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

A.2.3. The aop schema

The **aop** tags deal with configuring all things AOP in Spring. These tags are comprehensively covered in the chapter entitled <u>Chapter 13</u>, <u>Aspect Oriented Programming with Spring.NET</u>.

In the interest of completeness, to use the tags in the **aop** schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the **aop** namespace are available to you.

```
<!-- <object/> definitions here -->
 <!-- <aop/> AOP definitions here -->
</objects>
```

You will also need to configure the AOP namespace parser in the main .NET application configuration file as shown below



Note 🚫

As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

```
<configuration>
              <configSections>
                             <sectionGroup name="spring">
                                             <!-- other Spring config sections handler like context
                                           <section name="parsers" type="Spring.Context.Support.N</pre>
                                     </sectionGroup>
              </configSections>
              <spring>
                             <parsers>
                                            <parser type="Spring.Aop.Config.AopNamespaceParser, Spring.AopNamespaceParser, Spring.AopNamespac
                             </parsers>
              </spring>
</configuration>
```

A.2.4. The db schema

The db tags deal with creating IDbProvider instances for a given database client library. The following snippet references the correct schema so that the tags in the db namespace are available to you. The tags are comprehensively covered in the chapter entitled **Chapter 19**, **DbProvider**.

You will also need to configure the Database namespace parser in the main .NET application configuration file as shown below



Note 🔪

As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

A.2.5. The remoting schema

The remoting tags are for use when you want to export an existing POCO

object as a .NET remoted object or to create a client side .NET remoting proxy. The tags are comprehensively covered in the chapter Chapter 25, NET Remoting

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"</pre>
         xmlns:r="http://www.springframework.net/remoting">
 <!-- <object/> definitions here -->
  <!-- <r/> remoting definitions here -->
</objects>
```

You will also need to configure the remoting namespace parser in the main .NET application configuration file as shown below



As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <!-- other Spring config sections handler like context
      <section name="parsers" type="Spring.Context.Support.N</pre>
     </sectionGroup>
  </configSections>
  <spring>
    <parsers>
      <parser type="Spring.Remoting.Config.RemotingNamespace"
</pre>
    </parsers>
  </spring>
</configuration>
```

A.2.6. The NMS messaging schema

The nms tags are for use when you want to configure Spring's messaging support. The tags are comprehensively covered in the chapter Chapter 29, Message Oriented Middleware - Apache ActiveMQ

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"</pre>
         xmlns:r="http://www.springframework.net/nms">
 <!-- <object/> definitions here -->
  <!-- <nms/> remoting definitions here -->
</objects>
```

You will also need to configure the remoting namespace parser in the main .NET application configuration file as shown below



As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <!-- other Spring config sections handler like context
      <section name="parsers" type="Spring.Context.Support.N</pre>
     </sectionGroup>
  </configSections>
  <spring>
    <parsers>
      <parser type="Spring.Messaging.Nms.Config.NmsNamespace")</pre>
    </parsers>
  </spring>
```

A.2.7. The validation schema

The validation tags are for use when you want definte IValidator object instances. The tags are comprehensively covered in the chapter Chapter 12, Validation Framework

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"</pre>
        xmlns:v="http://www.springframework.net/validation">
 <!-- <object/> definitions here -->
  <!-- <v/> valdiation definitions here -->
</objects>
```

You will also need to configure the validation namespace parser in the main .NET application configuration file as shown below



As of Spring.NET 1.2.0 it is no longer necessary to explicitly configure the namespace parsers that come with Spring via a custom section in App.config. You will still need to register custom namespace parsers if you are writing your own.

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <!-- other Spring config sections handler like context
      <section name="parsers" type="Spring.Context.Support.N</pre>
     </sectionGroup>
  </configSections>
  <spring>
    <parsers>
      <parser type="Spring.Validation.Config.ValidationNames</pre>
```

```
/configuration>
```

A.2.8. The objects schema

Last but not least we have the tags in the <code>Objects</code> schema. Examples of the various tags in the <code>Objects</code> schema are not shown here because they are quite comprehensively covered in the section entitled Section 5.3.2, "Dependencies and configuration in detail" (and indeed in that entire Chapter).

A.3. Setting up your IDE

For SharpDevelop, follow the directions on the "<u>Editing XML</u>" product documentation.

Appendix B. Extensible XML authoring

B.1. Introduction

Spring supports adding custom schema-based extensions to the basic Spring XML format for defining and configuring objects. This section is devoted to detailing how you would go about writing your own custom XML object definition parsers and integrating such parsers into the Spring IoC container.

To facilitate the authoring of configuration files using a schema-aware XML editor, Spring's extensible XML configuration mechanism is based on XML Schema. If you are not familiar with Spring's current XML configuration extensions that come with the standard Spring distribution, please first read the appendix entitled Appendix A, XML Schema-based configuration.

Creating new XML configuration extensions can be done by following these (relatively) simple steps:

- 1. <u>Authoring</u> an XML schema to describe your custom element(s).
- 2. <u>Coding</u> a custom **INamespaceParser** implementation (this is an easy step, don't worry).
- 3. Coding one or more IObjectDefinitionParser implementations (this is where the real work is done).
- 4. Registering the above artifacts with Spring (this too is an easy step).

What follows is a description of each of these steps. For the example, we will create an XML extension (a custom XML element) that allows us to configure objects of the type Regex (from the System.Text.RegularExpressions namespace) in an easy manner. When we are done, we will be able to define object definitions of type Regex like this:

B.2. Authoring the schema

Creating an XML configuration extension for use with Spring's IoC container starts with authoring an XML Schema to describe the extension. What follows is the schema we'll use to configure Regex objects.

```
<?xml version="1.0" encoding="utf-8" ?>
<xsd:schema id="myns"</pre>
           xmlns="http://www.mycompany.com/schema/myns"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           xmlns:objects="http://www.springframework.net"
           xmlns:vs="http://schemas.microsoft.com/Visual-Stu
           targetNamespace="http://www.mycompany.com/schema/
           elementFormDefault="qualified"
           attributeFormDefault="unqualified"
           vs:friendlyname="Spring Regex Configuration" vs:i
           vs:iscasesensitive="true" vs:requireattributequot
           vs:defaultnamespacequalifier="" vs:defaultnsprefi
           >
  <xsd:import namespace="http://www.springframework.net"/>
  <xsd:element name="regex">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="objects:identifiedType">
          <xsd:attribute name="pattern" type="xsd:string" us</pre>
          <xsd:attribute name="options" type="xsd:string" us</pre>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

The emphasized line contains an extension base for all tags that will be identifiable (meaning they have an id attribute that will be used as the object identifier in the container). We are able to use this attribute because we imported the Spring-provided 'Objects' namespace. The VS: prefixed elements are for better integration with intellisense in VS.NET.

The above schema will be used to configure Regex objects, directly in an XML application context file using the <myns:regex/> element.

```
<myns:regex id="usZipCodeRegex"</pre>
             pattern="(^\d{5}$)|(^\d{5}-\d{4}$)"
             options="Compiled"/>
```

Note that after we've created the infrastructure classes, the above snippet of XML will essentially be exactly the same as the following XML snippet. In other words, we're just creating an object in the container, identified by the name 'usZipCodeRegex' of type Regex, with a couple of constructor arguments set.

```
<object id="usZipCodeRegex" type="System.Text.RegularExpre</pre>
  <constructor-arg name="pattern" value="(^\d{5}$)|(^\d{5})</pre>
  <constructor-arg name="options" value="Compiled"/>
</object>
```



The schema-based approach to creating configuration format allows for tight integration with an IDE that has a schema-aware XML editor. Using a properly authored schema, you can use intellisense to have a user choose between several configuration options defined in the enumeration. The schema for creating IDbProvider instances shows the use of XSD enumerations.

B.3. Coding a INamespaceParser

In addition to the schema, we need an <code>INamespaceParser</code> that will parse all elements of this specific namespace Spring encounters while parsing configuration files. The <code>INamespaceParser</code> should in our case take care of the <code>parsing</code> of the <code>myns:regex</code> element.

The **INamespaceParser** interface is pretty simple in that it features just two methods:

- Init() allows for initialization of the INamespaceParser and will be called by Spring before the handler is used
- IObjectDefinition Parse(Element, ParserContext) called when Spring encounters a top-level element (not nested inside a object definition or a different namespace). This method can register object definitions itself and/or return a object definition.

Although it is perfectly possible to code your own INamespaceParser for the entire namespace (and hence provide code that parses each and every element in the namespace), it is often the case that each top-level XML element in a Spring XML configuration file results in a single object definition (as in our case, where a single <myns:regex/> element results in a single Regex object definition). Spring features a number of convenience classes that support this scenario. In this example, we'll make the use NamespaceParserSupport class:

```
public override void Init()
{
          RegisterObjectDefinitionParser("regex", new Rege
      }
}
```

Notice that there isn't actually a whole lot of parsing logic in this class. Indeed... the NamespaceParserSupport class has a built in notion number delegation. It supports the registration of any of IObiectDefinitionParser instances, to which it will delegate to when it needs to parse an element in it's namespace. This clean separation of concerns allows an INamespaceParser to handle the orchestration of the parsing of all of the custom elements in it's namespace, while delegating to IObjectDefinitionParsers to do the grunt work of the XML parsing; this means that each IObjectDefinitionParser will contain just the logic for parsing a single custom element, as we can see in the next step.

To help in the registration of the parser for this namespace, the NamespaceParser attribute is used to map the XML namespace string, i.e. http://www.mycompany.com/schema/myns, to the location of the XML Schema file as an embedded assembly resource.

B.4. Coding an IObjectDefinitionParser

A IObjectDefinitionParser will be used if the INamespaceParser encounters an XML element of the type that has been mapped to the specific object definition parser (which is 'regex' in this case). In other words, the IObjectDefinitionParser is responsible for parsing *one* distinct top-level XML element defined in the schema. In the parser, we'll have access to the XML element (and thus it's subelements too) so that we can parse our custom XML content, as can be seen in the following example:

```
using System;
using System.Text.RegularExpressions;
using System.Xml;
using Spring.Objects.Factory.Support;
using Spring.Objects.Factory.Xml;
using Spring.Util;
namespace CustomNamespace
{
   public class RegexObjectDefinitionParser : AbstractSimple
   protected override Type GetObjectType(XmlElement element)
     return typeof (Regex); @
   }
   protected override void DoParse(XmlElement element, Object
       // this will never be null since the schema explicitl
       string pattern = element.GetAttribute("pattern");
       builder.AddConstructorArg(pattern);
       // this however is an optional property
       string options = element.GetAttribute("options");
       if (StringUtils.HasText(options))
       {
          RegexOptions regexOptions = (RegexOptions)Enum.Par
          builder.AddConstructorArg(regexOptions);
       }
```

```
protected override bool ShouldGenerateIdAsFallback
{
    get { return true; }
}
```

- We use the Spring-provided AbstractSingleObjectDefinitionParser to handle a lot of the basic grunt work of creating a *single* IObjectDefinition.
- We supply the AbstractSingleObjectDefinitionParser superclass with the type that our single IObjectDefinition will represent.

In this simple case, this is all that we need to do. The creation of our single IObjectDefinition is handled by the AbstractSingleObjectDefinitionParser superclass, as is the extraction and setting of the object definition's unique identifier. The property ShouldGenerateIdAsFallback will generate a throwaway object id incase one is not specified, this is useful when nesting object definitions.

B.5. Registering the handler and the schema

The coding is finished! All that remains to be done is to somehow make the Spring XML parsing infrastructure aware of our custom element; we do this by registering our custom <code>INamespaceParser</code> using a special configuration section handler. The location of the XML Schema in this example has been directly assoicated with the parser though the use of the <code>Namespace</code> attribute.

B.5.1. NamespaceParsersSectionHandler

The custom configuration section handler is of the type Spring.Context.Support.NamespaceParsersSectionH and is registered with .NET in the normal manner. The custom configuration section will simply point to the INamespaceParser implementation that has the Namespace attribute. For our example, we need to write the following:

B.6. Using a custom extension in your Spring XML configuration

Using a custom extension that you yourself have implemented is no different from using one of the 'custom' extensions that Spring provides straight out of the box. Find below an example of using the custom <regex/> element developed in the previous steps in a Spring XML configuration file.

B.7. Further Resources

Find below links to further resources concerning XML Schema and the extensible XML support described in this chapter.

- The XML Schema Part 1: Structures Second Edition
- The XML Schema Part 2: Datatypes Second Edition

Spring Framework



Appendix C. Spring.NET's spring-objects.xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns="http://www.springframework.net" xmlns:xs="</pre>
    <xs:annotation>
        <xs:documentation>
            Spring Objects XML Schema Definition
            Based on Spring Beans DTD, authored by Rod Johns
            Author: Griffin Caprio
            This defines a simple and consistent way of crea
            of managed objects configured by a Spring XmlObj
            This document type is used by most Spring functi
            web application contexts, which are based on obj
            Each object element in this document defines an
            Typically the object type (System. Type is specif
            object properties.
            Object instances can be "singletons" (shared ins
            (independent instances).
            References among objects are supported, i.e. set
            to refer to another object in the same factory of
            As alternative to object references, "inner obje
            Singleton flags and names of such "inner object"
            Inner object are anonymous prototypes.
            There is also support for lists, dictionaries, a
        </xs:documentation>
    </xs:annotation>
    <xs:annotation>
        <xs:documentation>Defines a base type for any requir
    </xs:annotation>
    <xs:simpleType name="nonNullString">
```

```
<xs:restriction base="xs:string">
        <xs:minLength value="0"/>
    </xs:restriction>
</xs:simpleType>
<xs:annotation>
    <xs:documentation>
        Element containing informative text describing t
        element. Always optional.
        Used primarily for user documentation of XML obj
    </xs:documentation>
</xs:annotation>
<xs:simpleType name="description">
    <xs:restriction base="nonNullString"/>
</xs:simpleType>
<xs:complexType name="valueObject">
    <xs:simpleContent>
        <xs:extension base="xs:string">
            <xs:attribute name="type" type="nonNullStrir</pre>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
<xs:complexType name="expression">
  <xs:sequence>
    <xs:element name="property" type="property" min0ccur</pre>
  </xs:sequence>
  <xs:attribute name="value" type="nonNullString" use="r</pre>
</xs:complexType>
<!--
     Defines a reference to another object in this factor
     factory (parent or included factory).
-->
<xs:complexType name="objectReference">
    <xs:attribute name="object" type="nonNullString" use</pre>
    <xs:attribute name="local" type="xs:IDREF" use="opti</pre>
    <xs:attribute name="parent" type="nonNullString" use</pre>
    <!--
         References must specify a name of the target of
         The "object" attribute can reference any name t
         to be checked at runtime.
         Local references, using the "local" attribute,
         they can be checked by this DTD, thus should be
         within the same object factory XML file.
```

```
-->
</xs:complexType>
<!-- Defines a reference to another object or a type.
<xs:complexType name="objectOrClassReference">
    <xs:attribute name="object" type="nonNullString" use</pre>
    <xs:attribute name="local" type="xs:IDREF" use="opti</pre>
    <xs:attribute name="type" type="nonNullString" use="</pre>
</xs:complexType>
<xs:group name="objectList">
    <xs:sequence>
        <xs:element name="description" type="description"</pre>
        <xs:choice>
            <xs:element name="object" type="vanillaObject"</pre>
            <!--
                  Defines a reference to another object i
                  factory (parent or included factory).
             -->
            <xs:element name="ref" type="objectReference"</pre>
            <!--
                  Defines a string property value, which
                  object in this factory or an external f
                  While a regular 'value' element could i
                  using idref in this case allows validat
                  parser, and name completion by helper t
            <xs:element name="idref" type="objectReferer</pre>
            <!--
                 A objectList can contain multiple inner
                  Lists are untyped, pending generics sup
                  strongly typed.
                 A objectList can also map to an array t
                  is automatically performed by Abstract(
            <xs:element name="list">
                 <xs:complexType>
                     <xs:group ref="objectList" min0ccurs</pre>
                     <xs:attribute name="element-type" ty</pre>
                 </xs:complexType>
            </xs:element>
            <!--
                 A set can contain multiple inner object
                  Sets are untyped, pending generics supp
```

```
strongly typed.
        -->
        <xs:element name="set">
            <xs:complexType>
                <xs:group ref="objectList" min0ccurs</pre>
            </xs:complexType>
        </xs:element>
        <!--
             A Spring map is a mapping from a string
             Maps may be empty.
        <xs:element name="dictionary" type="objectMa</pre>
        <!--
             Name-values elements differ from map el
             Name-values may be empty.
        -->
        <xs:element name="name-values" type="object"</pre>
        <!--
             Contains a string representation of a p
             The property may be a string, or may be
             required type using the System.Componer
             machinery. This makes it possible for a
             to write custom TypeConverter implement
             convert strings to objects.
             Note that this is recommended for simpl
             Configure more complex objects by setti
             to other objects.
        <xs:element name="value" type="value0bject"/</pre>
        <!--
             Contains a string representation of an
        <xs:element name="expression" type="expressi</pre>
        <!--
             Denotes a .NET null value. Necessary be
             will resolve to an empty String, which
             null value unless a special TypeConvert
        <xs:element name="null"/>
    </xs:choice>
</xs:sequence>
```

```
</xs:group>
<xs:complexType name="objectNameValues">
    <xs:sequence>
        <!--
              The "value" attribute is the string value of
              attribute is the name of the property.
        <xs:element name="add" min0ccurs="0" max0ccurs="</pre>
             <xs:complexType mixed="true">
                 <xs:attribute name="key" type="nonNullSt</pre>
                 <xs:attribute name="value" use="required"</pre>
                 <xs:attribute name="delimiters" use="opt</pre>
             </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="importElement">
    <xs:attribute name="resource" type="nonNullString" d</pre>
</xs:complexType>
<xs:complexType name="aliasElement">
    <xs:attribute name="name" type="nonNullString" use="</pre>
    <xs:attribute name="alias" type="nonNullString" use=</pre>
</xs:complexType>
<xs:complexType name="objectMap">
    <xs:sequence>
        <xs:element type="mapEntryElement" name="entry"</pre>
    </xs:sequence>
    <xs:attribute name="key-type" type="nonNullString" u</pre>
    <xs:attribute name="value-type" type="nonNullString"</pre>
</xs:complexType>
<xs:complexType name="mapEntryElement">
    <xs:sequence>
        <xs:element type="mapKeyElement" name="key" minC</pre>
        <xs:group ref="objectList" min0ccurs="0" max0ccu</pre>
    </xs:sequence>
    <xs:attribute name="key" type="nonNullString" use="d</pre>
    <xs:attribute name="value" type="nonNullString" use=</pre>
    <xs:attribute name="expression" type="nonNullString"</pre>
    <xs:attribute name="key-ref" type="nonNullString" us</pre>
    <xs:attribute name="value-ref" type="nonNullString"</pre>
</xs:complexType>
<xs:complexType name="mapKeyElement">
```

```
<xs:group ref="objectList" min0ccurs="1"/>
</xs:complexType>
<xs:annotation>
    <xs:documentation>Defines constructor argument.</xs:</pre>
</xs:annotation>
<xs:complexType name="lookupMethod">
    <xs:attribute name="name" type="nonNullString" use="</pre>
    <xs:attribute name="object" type="nonNullString" use</pre>
</xs:complexType>
<xs:complexType name="constructorArgument">
    <xs:group ref="objectList" min0ccurs="0"/>
    <1--
          The constructor-arg tag can have an optional r
          to specify a named parameter in the constructo
    <xs:attribute name="name" type="nonNullString" use="</pre>
    <!--
          The constructor-arg tag can have an optional i
          to specify the exact index in the constructor
          to avoid ambiguities, e.g. in case of 2 argume
    <xs:attribute name="index" type="nonNullString" use=</pre>
          The constructor-arg tag can have an optional t
          to specify the exact type of the constructor a
          to avoid ambiguities, e.g. in case of 2 single
          that can both be converted from a String.
    <xs:attribute name="type" type="nonNullString" use="</pre>
    <xs:attribute name="value" type="nonNullString" use=</pre>
    <xs:attribute name="expression" type="nonNullString"</pre>
    <xs:attribute name="ref" type="nonNullString" use="c</pre>
</xs:complexType>
<xs:annotation>
    <xs:documentation>Defines property.</xs:documentation</pre>
</xs:annotation>
<xs:complexType name="property">
    <xs:group ref="objectList" min0ccurs="0"/>
    <!-- The property name attribute is the name of the
    <xs:attribute name="name" type="nonNullString" use="</pre>
    <xs:attribute name="value" type="nonNullString" use=</pre>
    <xs:attribute name="expression" type="nonNullString"</pre>
```

```
<xs:attribute name="ref" type="nonNullString" use="d</pre>
</xs:complexType>
<xs:annotation>
    <xs:documentation>Defines a single named object.</xs</pre>
</xs:annotation>
<xs:complexType name="vanilla0bject">
    <xs:sequence>
        <xs:element name="description" type="description"</pre>
        <!--
              Object definitions can specify zero or more
              They correspond to either a specific index
              or are supposed to be matched generically t
              This is an alternative to "autowire constru
        <xs:element name="constructor-arg" type="constru"</pre>
              Object definitions can have zero or more pr
              Spring supports primitives, references to d
              related factories, lists, dictionaries and
        -->
        <xs:element name="property" type="property" min(</pre>
        <!--
              Object definitions can specify zero or more
        <xs:element name="lookup-method" type="lookupMet</pre>
        <!-- Object definitions can have zero or more re
        <xs:element name="replaced-method" min0ccurs="0"</pre>
             <xs:complexType>
                 <xs:sequence>
                     <xs:element name="arg-type" min0ccur</pre>
                           <xs:complexType>
                                <xs:attribute name="match"</pre>
                           </xs:complexType>
                     </xs:element>
                 </xs:sequence>
                 <xs:attribute name="name" type="nonNulls</pre>
                 <xs:attribute name="replacer" type="non"</pre>
            </xs:complexType>
        </xs:element>
        <!-- Object definitions can have zero or more su
        <xs:element name="listener" minOccurs="0" maxOcc</pre>
            <xs:complexType>
```

```
<xs:sequence>
                <xs:element name="ref" type="object()</pre>
            </xs:sequence>
            <!-- The event(s) the object is interest
            <xs:attribute name="event" type="nonNull</pre>
            <!-- The name or name pattern of the met
            <xs:attribute name="method" type="nonNul</pre>
        </xs:complexType>
    </xs:element>
</xs:sequence>
<!--
     Objects can be identified by an id, to enable r
     There are constraints on a valid XML id: if you
     in .NET code using a name that's illegal as an
     "name" attribute. If neither given, the object
-->
<xs:attribute name="id" type="xs:ID" use="optional"/</pre>
<!--
     Optional. Can be used to create one or more ali
     Multiple aliases can be separated by any number
<xs:attribute name="name" type="nonNullString" use="</pre>
<!--
     Each object definition must specify the full, a
     or the name of the parent object from which the
     Note that a child object definition that refere
     add respectively override property values and b
     singleton status. It will inherit all of the pa
     like lazy initialization or autowire settings.
-->
<xs:attribute name="type" type="nonNullString" use="</pre>
<xs:attribute name="parent" type="nonNullString" use</pre>
<!--
     Is this object "abstract", i.e. not meant to be
     rather just serving as parent for concrete chil
     Default is false. Specify true to tell the obje
     instantiate that particular object in any case.
<xs:attribute name="abstract" type="xs:boolean" use=</pre>
<!--
     Is this object a "singleton" (one shared instar
```

be returned by all calls to GetObject() with the or a "prototype" (independent instance resulting getObject(). Default is singleton.

Singletons are most commonly used, and are idea service objects.

-->

<xs:attribute name="singleton" type="xs:boolean" use
</pre>/--

Optional attribute controlling the scope of single only applicable to ASP. Net web applications and objects. Applications other than ASP. Net web applications of the standard or services and the standard or services.

1. "application"

Default object scope. Objects defined with appl traditional singleton objects. Same instance wi to IApplicationContext.GetObject()

2. "session"

Objects with this scope will be stored within used typically used for objects such as shopping

3. "request"

Object with this scope will be initialized for objects, same instance will be returned from al within the same HTTP request. For example, if a Server. Transfer method, they can easily share t request-scoped object.

-->

Is this object to be lazily initialized?
If false, it will get instantiated on startup to that perform eager initialization of singletons

Optional attribute controlling whether to "auto This is an automagical process in which object explicitly in the XML object definition file, k

There are 5 modes:

1. "no"

The traditional Spring default. No automagical must be defined in the XML file via the <ref> e in most cases as it makes documentation more ex

2. "byName"

Autowiring by property name. If a object of cla Spring will try to set this to the value of the

3. "byType"

Autowiring if there is exactly one object of the If there is more than one, a fatal error is rai autowiring for that object. If there is none, redependency-check="objects" to raise an error in

4. "constructor"

Analogous to "byType" for constructor arguments of the constructor argument type in the object

5. "autodetect"

Chooses "constructor" or "byType" through intro If a default constructor is found, "byType" get

The latter two are similar to PicoContainer and configure for small namespaces, but doesn't wor

```
behaviour for bigger applications.
     Note that explicit dependencies, i.e. "property
     always override autowiring. Autowire behaviour
     checking, which will be performed after all aut
-->
<xs:attribute name="autowire" use="optional" default</pre>
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="no"/>
            <xs:enumeration value="byName"/>
            <xs:enumeration value="byType"/>
            <xs:enumeration value="constructor"/>
            <xs:enumeration value="autodetect"/>
            <xs:enumeration value="default"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
<!--
     Optional attribute controlling whether to check
     objects dependencies, expressed in its properti
     Default is no dependency checking.
     "simple" type dependency checking includes prin
     "object" includes collaborators (other objects
     "all" includes both types of dependency checking
-->
<xs:attribute name="dependency-check" use="optional"</pre>
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="none"/>
            <xs:enumeration value="objects"/>
            <xs:enumeration value="simple"/>
            <xs:enumeration value="all"/>
            <xs:enumeration value="default"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
<!--
     The names of the objects that this object deper
     The object factory will quarantee that these of
```

```
Note that dependencies are normally expressed t
         constructor arguments. This property should jus
         of dependencies like statics (*ugh*) or databas
    <xs:attribute name="depends-on" type="nonNullString"</pre>
    <!--
         Optional attribute for the name of the custom i
         to invoke after setting object properties. The
         but may throw any exception.
    <xs:attribute name="init-method" type="nonNullString</pre>
    <1--
         Optional attribute for the name of the custom d
         on object factory shutdown. The method must hav
         but may throw any exception. Note: Only invoked
    -->
    <xs:attribute name="destroy-method" type="nonNullStr</pre>
    <xs:attribute name="factory-method" type="nonNullStr</pre>
    <xs:attribute name="factory-object" type="nonNullStr</pre>
</xs:complexType>
<xs:annotation>
    <xs:documentation>The document root. At least one d
</xs:annotation>
<xs:element name="objects">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="description" type="descrip"</pre>
            <xs:choice min0ccurs="0" max0ccurs="unbounde"</pre>
                 <xs:element name="import" type="importEl</pre>
                 <xs:element name="alias" type="aliasElem</pre>
                 <xs:element name="object" type="vanilla0"</pre>
                 <xs:any namespace="##other" processConte</pre>
            </xs:choice>
        </xs:sequence>
        <!--
              Default values for all object definitions.
              the "object" level. See those attribute def
        -->
        <xs:attribute name="default-lazy-init" type="xs:</pre>
        <xs:attribute name="default-dependency-check" us</pre>
            <xs:simpleType>
```

```
<xs:restriction base="xs:string">
                         <xs:enumeration value="none"/>
                         <xs:enumeration value="objects"/>
                         <xs:enumeration value="simple"/>
                         <xs:enumeration value="all"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="default-autowire" use="optic"</pre>
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                         <xs:enumeration value="no"/>
                         <xs:enumeration value="byName"/>
                         <xs:enumeration value="byType"/>
                         <xs:enumeration value="constructor"/</pre>
                         <xs:enumeration value="autodetect"/>
                     </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
        </xs:complexType>
    </xs:element>
</xs:schema>
```