Deployment Tools Foundation **Deployment Tools Foundation** Overview v4.0

Deployment Tools Foundation is a rich set of .NET class libraries and related resources that together bring the Windows deployment platform technologies into the .NET world. It is designed to greatly simplify deployment-related development tasks while still exposing the complete functionality of the underlying technology.

The primary focus of DTF is to provide a foundation for development of various kinds of tools to support deployment throughout the product lifecycle, including setup authoring, building, analysis, debugging, and testing tools. In addition to tools, DTF can also be useful for install-time activities such as setup bootstrappers, external UI, and custom actions, and for application run-time activities that need to access the deployment platform.

For a description of the the latest changes, see What's New.

Highlights

- New project name name "Deployment Tools Foundation", and new namespaces Microsoft.Deployment.*
- Added ZIP compression library
- Added library for reading/writing Win32 resources including file versions
- Managed custom action improvements:
 - Simplified authoring and building -- new MakeSfxCA tool automatically maps DLL entrypoints to CA methods.
 - Managed custom action DLLs now run in a separate process for better reliability with respect to CLR versions, but still have full access to the MSI session.
- Found and fixed many bugs with extensive unit test suite
- LINQ to MSI ! (preview)

Unfortunately, all these changes do mean that migrating tools and applications from the previous release can be a moderate amount of work.

Breaking Changes

For the first time since v1.0, this release contains some major breaking changes, due to a significant redesign and cleanup effort that has been long time coming. The overall purpose of the changes is to bring the class libraries much closer to ship-quality.

- All libraries use a new namespace hierarchy under Microsoft.Deployment. The new namespace aligns with the new project name, gives all the various libraries an identity that makes them obviously related to the DTF project, and mostly avoids "takin over" a namespace that might be rightfully owned by the platform technology owner.
- Assemblies are also renamed to follow namespaces.
- A new unified compression framework forms the basis for the new ZIP library and a redesigned CAB library. Additional archive format: can be plugged into the framework. The stream-based compression APIs have been redesigned to be more object-oriented and easier use. The file-based APIs are mostly unchanged from the old cabine library, although some names have changed in order to fit into the new unified framework.
- Large parts of the WindowsInstaller library have been redesigned to be more object-oriented and to better follow .NET Framework desig guidelines. And various APIs throughout the library have naming or other changes for better consistency and to follow conventions and best pratices as enforced by FxCop.
- The WindowsInstaller APIs no longer make any attempt to mimic th MSI COM automation interfaces. The naming and object patterns in the automation interfaces often conflicted with with best practices for .NET Framework class libraries. Since most people start using DTF without having ever experienced MSI scripting, there is little reason to match the scripting object model. Making the APIs more consistent with .NET conventions will make them much easier to us for people already experienced with the .NET Framework.
- APIs in all class libraries use generics where appropriate, especial the generic collection interfaces. This means .NET Framework 2.0 later is required.
- The FilePatch library is missing from this release. An updated and redesigned file-delta library is in development.

Other Changes

- New MakeSfxCA tool for building managed custom action package In addition to packaging the CA DLL and dependencies, it automatically detects managed CA methods and generates corresponding unmanaged DLL entrypoints in the CA host DLL (SfxCA.dll), where they are called by MSI. Previously it was necessary to either provide this mapping in a CustomAction.config file, or live with the generic "ManagedCustomActionN" names wher authoring the CustomAction table in the MSI. For more info, see the help topic on building managed custom actions.
- Out-of-proc managed custom action DLLs: When a managed custo action runs, it normally requests a specific major version of the CLF via CustomAction.config. However in the previous implementation, the request could be ignored if there was already a different versior of the CLR loaded into the MSI process, either from a previous custom action or by some other means. (The CLR doesn't allow side-by-side versions within the same process.) While there have been no reports of this issue causing setup failures in practice, it may be only a matter of time, as new CLR versions keep coming or

The redesigned native host for managed custom actions, SfxCA.dll re-launches itself in a separate process before loading the CLR and invoking the managed CA. This ensures that the desired CLR version is always loaded, assuming it is available on the system. It also sets up a named-pipe remoting channel between the two processes. All session-related MSI API calls are routed through tha channel, so that the custom action has full access to the installer session just as if it were running in-process.

• The new zip compression library supports nearly all features of the zip file format. This includes the ZIP64 extensions for archives greater than 4GB, as well as disk-spanning capabilities. Zip encryption is not supported. The zip library has been tested against a variety of third-party zip tools; please report any issues with incompatible packages.

Currently only the basic DEFLATE compression algorithm is supported (via System.IO.Compression.DeflateStream), and the compression level is not adjustable when packing an archive. The zip file format has a mechanism for plugging in arbitrary compression algorithms, and that capability is exposed: you can provide a Streat object capable of compressing and decompressing bytes as an alternative to DeflateStream.

- Added support for the few APIs new in MSI 4.0:
 - Installer.GetPatchFileList()
 - InstallLogModes.RMFilesInUse
 - ComponentAttributes.DisableRegistryReflection
 - ControlAttributes.ElevationShield
- The documentation is now built with the **Sandcastle** doc build engine, with help from the **Sandcastle Help File Builder**. (The old CHM was built with NDoc.)
- The documentation includes detailed class diagrams for the WindowsInstaller and Compression namespaces.
- WindowsInstaller API doc topics now link straight to the corresponding unmanaged MSI API topics in MSDN. If you know a unmanaged MSI API you want to use but don't know the managed equivalent, you can search for it and find what managed APIs link t it.
- Unit tests cover about 90% of the Compression, Compression.Zip, and Compression.Cab assemblies -- basically everything except some rare error-handling cases.
- Unit tests along with samples cover over 50% of the WindowsInstaller and WindowsInstaller.Package assemblies (including custom action functionality). More test cases are still beir added.

Bugfixes

In addition to the extensive cleanup due to redesigns and unit tests, the following reported bugs have been fixed:

- Managed custom actions could in rare instances fail to load with error 183 (directory already exists)
- Timestamps of files in a cabinet could be incorrectly offset based of the timezone. (This was due to a behavior change in the DateTime class between .NET 1.1 and 2.0.)
- Unicode file paths for cabbed files could be handled incorrectly in some cases
- Installer.DetermineApplicablePatches just didn't work
- InstallPackage.ApplyPatch couldn't handle applying multiple patche to the same layout

LINQ to MSI

You'll never want to write MSI SQL again!

Language INtegrated Query is a new feature in .NET Framework 3.5 an C# 3.0. Through a combination of intuitive language syntax and powerfu query operations, LINQ provides a whole new level of productivity for working with data in your code. While the .NET Framework 3.5 provides LINQ capability for SQL databases and XML data, now you can write LINQ queries to fetch and even update data in MSI databases!

Look at the following example:

```
var actions = from a in db.InstallExecuteSequences
    join ca in db.CustomActions on a.Action equals ca.Action
    where ca.Type == CustomActionTypes.Dll
    orderby a.Sequence
    select new {
        Name = a.Action,
        Target = ca.Target,
        Sequence = a.Sequence };
```

```
foreach (var a in actions)
{
    Console.WriteLine(a);
}
```

The query above gets automatically translated to MSI SQL:

```
SELECT `InstallExecuteSequence`.`Action`, `CustomAction`.`Target`,
`InstallExecuteSequence`.`Sequence` FROM `InstallExecuteSequence`,
`CustomAction` WHERE `InstallExecuteSequence`.Action` =
`CustomAction`.`Action` ORDER BY `InstallExecuteSequence`.`Sequence`
```

But the query is not executed until the foreach enumeration. Then recorc are fetched from the results incrementally as the enumeration progresses. The objects fetched are actually of an anonymous type created there in the query with exactly the desired fields. So the result c this code will be to print the Action, Target, and Sequence of all Type 1 custom actions.

The query functionality is currently limited by the capabilities of the MSI SQL engine. For example, a query can't use where (ca.Type & CustomActionTypes.Dll) != 0 because the bitwise-and operator is not supported by MSI SQL. The preview version of LINQ to MSI will throw ϵ exception for cases like that, but the eventual goal is to have it automatically move the data and operation outside of MSI when necessary, so that any arbitrary expressions are supported in the query.

Note there are no MSI handles (or IDisposables) to worry about! Handles are all managed internally and closed deterministically. Also, with the entity object model for common tables, the compiler will tell you if you get a column name wrong or misspelled. The entity objects even support easy inserting, updating, and deleting (not shown here).

For more examples, see the LinqTest project in the source. More documentation is being written.

Obviously, LINQ to MSI requires .NET Framework 3.5. Everything else i

DTF requires only .NET Framework 2.0.

Note: The LINQ functionality in this DTF release is of preview quality on and should not be used in production. While there are unit tests coverin a wide variety of queries, using advanced queries outside what is covered by the tests is likely to result in unexpected exceptions, and retrieved data might possibly be incorrect or incomplete. An updated LINQ to MSI library is in development.

Deployment Tools Foundation Change History Overview > Change History

2007-07-03

See What's New?

2005-03-30

- New custom action proxy
 - Managed custom actions use an XML config file to specify the CLR version.
 - New CAPack module is an unmanaged self-extracting CA DLL that can wrap both managed and unmanaged custom actions. (The old managed CAProxy module is obsolete.)
 - Custom action build process is different but still complicated -- see documentation for details.
 - CustomActionAttribute no longer accepts the optional NativeDependencies parameter since it does not apply to the new proxy (all packaged files are always extracted and availab when the CA executes).
- 64bit support
 - Various code fixes to pointer/handle types and structure alignments.
 - Cabinet and MSI libraries tested on AMD64 CLR.
 - Unmanaged and managed parts of custom action proxy tested on AMD64.
- MSI 3.1 APIs added:
 - Installer.SetExternalUI(ExternalUIRecordHandler)
 - Installer.NotifySidChange
- Code builds easier with .NET Famework 2.0
 - AugmentIL post-build step is no longer necessary when compiling the cabinet code against .NET Framework 2.0, whicl

has builtin support for cdecl delegates.

- All C# code compiles against .NET Framework 2.0 without obsolete warnings, when setting the NETFX2 preprocessor define.
- Same code is still compatible with .NET Framework 1.0 + AugmentIL.
- Miscellaneous bugfixes/changes:
 - InstallPackage.ExtractFiles could fail in some cominations of compressed/uncompressed files fixed.
 - Installer.DeterminePatchSequence was broken due to an incorrect interop struct - fixed.
 - CabinetInfo and CabinetFileInfo classes made serializable.
 - Added Session.FormatString method to simplify formatting a string with property substitutions.
- Documentation updates:
 - Updated all documentation for new CA proxy.
 - Added new topic discussing InstallUtil.

2004-04-13

- Documentation
 - Consolidated all documentation into a single CHM file.
 - Added new topics about working with MSI databases & cabine files, to help new users get oriented more easily.
- WindowsInstaller
 - Removed [Beta] tags from MSI 3.0 APIs, but otherwise there have been no changes since 3.0 Beta 1.
 - Be warned these are still the least-tested parts of the library, so early users may encounter bugs.
- InstallPackage
 - Fixed InstallPackage.ExtractFiles() bug when directory doesn't exist.
 - Added ability to handle uncompressed files in a package marked as compressed.
- Cabinet
 - Fixed improper handling of file attributes.

- This bug caused some packages to not be extractable by other tools.
- Added support for UTF filenames.
 - Non-ASCII filenames will automatically be stored as UTF-(But note most other tools don't know how to extract them

2003-10-13

- Cab
 - Fixed a bug introduced in v2.4.0 that caused files to be left in the %TEMP% directory after creating a cab.
 - Unsealed the CabinetInfo, CabinetFileInfo, CabinetStatus classes and made a few methods protected and virtual.
- AugmentIL
 - Fixed a bug that sometimes caused a crash when specifying a relative output path on the command-line.
 - Fixed a bug that sometimes caused the Win32 version to be missing from the output file.
- Samples\Diff: added new sample tool
 - Recursively diffs directories, MSIs, MSPs, CABs, other files.

2003-09-23

- Cab
 - Fixed a bug that caused compressing very large files/file sets t use way too much memory. Performance on large inputs is nov within a few % of native cab tools (sometimes even a little faster!) for the same compression level.
- WindowsInstaller
 - All the new MSI 3.0 beta APIs are wrapped, resulting in the following additions:
 - New classes Product, Patch (for accessing sourcelist an other config)
 - New methods on Install class GetProducts, GetPatches, RemovePatches, ApplyMultiplePatches,

DetermineApplicablePatches, ExtractPatchXmlData

- New enumerations InstallContext, PatchStates, SourceType
- Additional InstallProperty values
- Note, MSI 3.0 support should be considered preliminary for no as APIs (both native and managed) are subject to change.
- For MSI 2.0 compatibility, developers should not use any classes or methods that are marked as [MSI 3.0 beta] in the documentation.
- And unrelated to 3.0, a few additional enums have been addec DialogAttributes, ControlAttributes, CustomActionTypes, IniFileAction, RegistryRoot, RemoveFileInstallMode, ServiceControlEvents, ServiceInstallFlags, TextStyles, UpgradeAttributes, LocatorType
- Also made a few minor non-breaking changes to keep the library FxCop-clean.
- AugmentIL
 - Added support for strongname signing and delay-signing. AugmentIL tries to locate the keyfile using the AssemblyKeyFileAttribute, or you may specify the path with the new /key option.
 - All "released" assemblies will now be strongname-signed (with an unofficial key).
- CAProxy
 - Added support for NativeDependencies property on CustomActionAttribute. This enables custom actions to P/Invol into native DLLs that are carried with them.
- Samples\SampleCAs
 - In SampleCA2, changed MessageBox.Show("") to session.Message(User,""), because generally it is a bad practic for CAs to show independent UI.
 - Added test of CustomActionAttribute.NativeDependencies functionality.
- Samples\CabPack: added new sample
 - Demonstrates & tests the cab library by creating self-extracting packages
- Samples\Inventory: added new sample

 Shows a hierarchical, relational, searchable view of all of the product, feature, component, file, and patch data managed by MSI, for all products installed on the system.

2003-09-12

- Cab:
 - Added CabinetInfo.CompressDirectory method, capable of compressing an entire directory tree structure.
 - Updated documentation of various methods concerning suppo of directory structure inside cabs.
 - CabinetInfo case-sensitivity was inconsistent now it is caseinsensitive by default, though case is still preserved
 - Separated assembly attributes into assembly.cs
- Msi:
 - InstallerException and subclasses automatically get extended error data from MSI's last-error-record when available. The dat is stored in the exception and made available through the GetErrorRecord() method, and the exception's Message includes the formatted error message and data. This makes most exceptions extremely informative!
 - Added View.GetValidationErrors() method, and supporting ValidationErrorInfo struct and ValidationError enum. This wrapper for the MsiViewGetError API had been accidentally let out.
 - Session.Message() now supports message-box flags to specify buttons & icon
 - Added doc remarks to various methods about closing handles.
 - Separated assembly attributes into assembly.cs
- AugmentIL:
 - Recent builds of ildasm v2.0.* have a slightly different output format, which could break AugmentIL in some cases fixed
- SampleCAs:
 - Removed 'using' clause from SampleCA1 -- there's no need to close the session's active database handle
- Documentation:

 Added note to ReadMe about compiling the cab source into another assembly

2003-08-07

- Cab:
 - CabinetInfo.IsValid() usually returned false even for valid cabs fixed
 - Extracting cab files with null timestamps generated exception fixed
- Msi:
 - Added InstallCanceledException, subclass of InstallerExceptio Methods which may be canceled by the user can throw this exception
 - Added MessageResult enumeration; Used by Session.Message() and ExternalUIHandler delegate
 - Installer.EnableLog() now supports extended attributes correct Append mode and flush-every-line
 - Added Session.DoActionSequence() This wrapper for the MsiSequence API had been accidentally left out
- CAProxy:
 - Catches InstallCanceledException, returns ERROR_INSTALL_USEREXIT so CA developer doesn't necessarily have to handle the exception
- Msi\Package:
 - Added TransformInfo class: metadata about an individual patcl transform
 - Added PatchPackage.GetTransform*() methods which return TransformInfo
- Documentation:
 - Added section to ReadMe.htm about building managed custon actions

2003-06-02

- Msi:
 - Validation didn't work on merge modules fixed
- CAProxy:
 - Was broken in 2.1 fixed

2003-05-14

- Msi:
 - External UI handler didn't survive a garbage collection fixed
 - Validation engine was completely broken now it should work a least for MSIs which are already mostly valid
 - Added DynamicLoad property to CustomActionAttribute Usage: set DynamicLoad=false when using XmlSerialization; default is true
- Msi\Package:
 - File extraction and update methods didn't work on merge modules - fixed
 - Made file update code slightly more robust
 - Removed hard-reference to the FilePatch assembly now it is only loaded if working with binary file patches
- AugmentIL:
 - AugmentIL would crash if some input files had read-only attr fixed
 - Made /verbose switch slightly more verbose
- CAProxy:
 - Added support for the DynamicLoad property of CustomActionAttribute
 - Added MMsiBreak debugging functionality see doc
- Samples\WiFile:
 - Added /I (list files) switch
- Samples\SampleCAs:
 - In the makefile the comments about debug builds had an error Now the sample builds debug packages (correctly) by default.
- Documentation:
 - Wrote AugmentIL.htm describing the AugmentIL tool and its options.

- Wrote WiFile.htm describing the WiFile sample tool.
- Added section to ReadMe.htm about debugging managed custom actions.

2003-03-31

- Msi: Implemented the remaining APIs, also minor improvements ar bugfixes
 - All published APIs are wrapped, with the exception of four: MsiGetFileSignatureInformation (because I don't know of a .NE analog for the returned certificate structure), and 3 APIs for previewing UI
 - Database.OpenView and Database.Execute* now take String.Format style params
 - Database.ApplyTransform can optionally use the errorsuppression flags stored in the transform summary info
 - Added a few supporting enumerations and structures for the remaining APIs
 - InstallerException gets a descriptive message for any MSI or system error
 - Fixed a bug in InstallerException which would usually report "error 0"
 - Added optimization for setting a Record field to a MemoryStream
 - Record.GetStream is capable of extracting substorages
 - Moved InstallPath class to Microsoft.WindowsInstaller.Package assembly
- Msi\FilePatch: added new project
 - Binary file patch API wrapper
- Msi\Package: added new project
 - $\circ~$ Helper classes for working with MSI and MSP packages
- Cab: some minor bugfixes
 - Cabinet.Extract(stream, name) threw a NullReferenceExceptio if the file didn't exist in the cab -- now it returns null
 - CabinetInfo.CompressFileSet() was broken -- fixed
 - If a Cabinet callback throws an exception, it is propogated as

the inner-exception of the CabinetException

- Samples\WiFile: added new sample
 - Demonstrates some features of InstallPackage class in Msi\Package project

2003-03-20

Documentation!

- Msi and Cab sources include complete C# XML documentation.
- Msi and Cab makefiles generate XML documentation files.
- Reference CHM compiled from XML documentation with NDoc.

I am aware that exceptions are still not documented in most areas. Othe than that, feel free to send me a note if it's still not clear how to use part of the API after reading the documentation.

Version is still 1.1 because there are no code changes in this release.

2003-03-13

- Msi: lots of small improvements for usability and consistency
 - Reworked ExternalUIHandler support
 - Added Installer properties/methods:
 - Components
 - ComponentClients()
 - ComponentState()
 - ComponentPath()
 - EnableLog()
 - Added Session.EvaluateCondition() method
 - Improved exception-handling in many methods in Installer, Database, & Session classes
 - Added extensive XML doc-comments to Installer, Database, View, & Session classes
 - A few breaking changes:
 - View.ModifyMode enumeration moved outside View and

renamed ViewModifyMode

- InstallLogMode enumeration renamed to InstallLogModes (naming convention for bitfields)
- Record constructor takes arbitrary number of parameters
- AugmentIL: almost completely rewritten
 - Ildasm/ilasm steps are built-in
 - The round-trip can be done in one step
 - IL source input/output is still supported
 - Never throws an unhandled exception
 - Organized command-line options, consistent with other .NET tools
 - Uses a plugin architecture to allow additional augmentations
- CAProxy: Added AIL_CAProxy.cs AugmentIL plugin generates C/ proxy methods
- SampleCAs: Updated makefile for new AugmentIL usage

2003-01-16

- ReadMe.htm: Added section on writing managed CAs
- SampleCAs: Added missing reference to System.Windows.Forms.(to the makefile
- AugmentIL: Added specific warning messages for when CA method has wrong signature
- Put sources in Toolbox-hosted Source Depot.

2003-01-14

Initial posting to http://toolbox

This page lists all the components that the DTF project depends on, at build time and at run-time.

Build-time Dependencies

- Visual Studio / .NET Framework Most of DTF can be built with Visual Studio 2005 & .NET Framework 2.0. However, the LINQ project requires VS 2008 & .NET Framework 3.5.
- Sandcastle .NET documentation build engine from Microsoft, use to process all the XML doc-comments in DTF libraries into DTFAPI.chm. (official site)
- Sandcastle Builder Sandcastle by itself is complex and difficult to use; this free tool from Codeplex provides an easy-to-use project system around it to automate the documentation build process. (project link)
- **HTML Help Workshop** Tools for building HTML Help 1.x (CHM files). Used to build DTF.chm. (download link)

Run-time Dependencies

- **.NET Framework** Most of DTF requires .NET Framework 2.0. (.NET 1.1 is no longer supported.) The only exception is the LINQ assembly which requires .NET Framework 3.5.
- Windows Installer Windows Installer introduced new APIs and capabilities with each successive version. Obviously, the corresponding functionality in the managed APIs is only available when the required version of the Windows Instaler (msi.dll) is installed on the system. Use the Installer.Version property to easily check the currently installed MSI version. Attempting to use an API not supported by the current version will result in an

EntryPointNotFoundException. To check what version is required for a particular API, see the documentation link to the corresponding unmanaged API in MSI.chm.

In some instances when a newer version of MSI provides an "Ex" alternative to a function, only the "Ex" function is used by the managed library. This may hide some functionality that would have otherwise been available on a system with an older version of MSI.

- **cabinet.dll** The DTF cabinet compression library uses cabinet.dll implement the low-level cabinet compression and decompression. This DLL is part of all versions of Windows, located in the system directory.
- System.IO.Compression.DeflateStream The DTF zip compression library uses this class to implement the low-level zip compression and decompression. This class is part of .NET Framework 2.0 and later.

Deployment Tools Foundation Support/Bugs Overview > Support/Bugs

Please send general support questions or comments to the **wix-users** discussion list.

Bugs, suggestions, or feature requests can be submitted at the **WiX project** on Sourceforge.net.

Deployment Tools Foundation Deployment Tools Foundation Development Guide

Development Guide

- Managed Custom Actions
- Working with MSI Databases
- Working with Cabinet Files
- Working with Install Packages
- Sample Applications

Deployment Tools Foundation Managed Custom Actions
Development Guide > Managed CAs

- Writing Managed Custom Actions
- Specifying the Runtime Version
- Building Managed Custom Actions
- Debugging Managed Custom Actions
- About InstallUtil

Deployment Tools Foundation Writing Managed Custom

Actions

Development Guide > Managed CAs > Writing CAs

Caveats

Before choosing to write a custom action in managed code instead of traditional native C++ code, you should carefully consider the following:

- Obviously, it introduces a dependency on the .NET Framework. You MSI package should probably have a LaunchCondition to check for the presence of the correct version of the .NET Framework before anything else happens.
- If the custom action runs at uninstall time, then even the uninstall o your product may fail if the .NET Framework is not present. This means a user could run into a problem if they uninstall the .NET Framework before your product.
- A managed custom action should be configured to run against a specific version of the .NET Framework, and that version should match the version your actual product runs against. Allowing the version to "float" to the latest installed .NET Framework is likely to lead to compatibility problems with future versions. The .NET Framework provides side-by-side functionality for good reason -- us it.

How To

• A custom action function needs to be declared as public static (aka Public Shared in VB.NET). It takes one parameter which is a Session object, and returns a ActionResult enumeration.

[CustomAction] public static ActionResult MyCustomAction(Session session)

- The function must have a **CustomActionAttribute**, which enables to be linked to a proxy function. The attribute can take an optional "name" parameter, which is the name of the entrypoint that is exported from the custom action DLL.
- Fill in MSI CustomAction table entries just like you would for a normal type 1 native-DLL CA. Managed CAs can also work just as well in deferred, rollback, and commit modes.
- If the custom action function throws any kind of Exception that isn't handled internally, then it will be caught by the proxy function. The Exception message and stack trace will be printed to the MSI log if logging is enabled, and the CA will return a failure code.
- To be technically correct, any MSI handles obtained should be closed before a custom action function exits -- otherwise a warning gets printed to the log. The handle classes in the managed library (Database, View, Record, SummaryInfo) all implement the IDisposable interface, which makes them easily managed with C#'s using statement. Alternatively, they can be closed in a finally block. As a general rule, methods return new handle objects that should b managed and closed by the user code, while properties only return reference to a prexisting handle object.
- Don't forget to use a **CustomAction.config** file to specify what version of the .NET Framework the custom action should run against.

See also:

- Sample C# Custom Actions
- Specifying the Runtime Version
- Working with MSI Databases
- Building Managed Custom Actions
- Debugging Managed Custom Actions

Deployment Tools Foundation Specifying the Runtime Version Development Guide > Managed CAs > Writing CAs > CustomAction.config

Every managed custom action package should contain a CustomAction.config file, even though it is not required by the toolset. Here is a sample:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<startup>
<supportedRuntime version="v2.0.50727"/>
</startup>
</configuration>
```

The configuration file follows the standard schema for .NET Framework configuration files **documented on MSDN**.

Supported Runtime Version

In the startup section, use **supportedRuntime** tags to explicitly specify the version(s) of the .NET Framework that the custom action should run on. If no versions are specified, the chosen version of the .NET Framework will be the "best" match to what Microsoft.Deployment.WindowsInstaller.dll was built against.

Warning: leaving the version unspecified is dangerous as it introduces a risk of compatibility problems with future versions of the .NET Framework. It is highly recommended that you specify only the version(s) of the .NET Framework that you have tested against.

Other Configuration

Various other kinds of configuration settings may also be added to this file, as it is a standard **.NET Framework application config file** for the custom action.

See also:

- Writing Managed Custom Actions
- Building Managed Custom Actions
- Proxy for Managed Custom Actions

Deployment Tools Foundation Sample C# Custom Action Development Guide > Managed CAs > Writing CAs > C# Sample

MSI custom actions are MUCH easier to write in C# than in C++!

```
[CustomAction]
public static ActionResult SampleCustomAction1(Session session)
{
  session.Log("Hello from SampleCA1");
  string testProp = session["SampleCATest"];
  string testProp2;
  testProp2 = (string) session.Database.ExecuteScalar(
    "SELECT `Value` FROM `Property` WHERE `Property` = 'SampleCA'
  if(testProp == testProp2)
  {
    session.Log("Simple property test passed.");
    return ActionResult.Success:
  }
  else
    return ActionResult.Failure;
}
```

A sample CA project with two CAs is included in the

Samples\ManagedCA directory. Running the CustomActionTest project will package the CA and insert it into a test MSI. The MSI will invoke the custom actions, but it will not install anything since the second sample (returns ActionResult.UserExit.

See also:

- Writing Managed Custom Actions
- Specifying the Runtime Version
- Working with MSI Databases
- Building Managed Custom Actions

• Debugging Managed Custom Actions

Send comments on this topic to wix-users@lists.sourceforge.net

Deployment Tools Foundation Building Managed Custom

Actions

Development Guide > Managed CAs > Building

The build process for managed CA DLLs is a little complicated becuase of the proxy-wrapper and dll-export requirements. Here's an overview:

- 1. Compile your CA assembly, which references Microsoft.Deployment.WindowsInstaller.dll and marks exported custom actions with a CustomActionAttribute.
- Package the CA assembly, CustomAction.config, Microsoft.Deployment.WindowsInstaller.dll, and any other dependencies using MakeSfxCA.exe. The filenames of CustomAction.config and Microsoft.Deployment.WindowsInstaller.d must not be changed, since the custom action proxy specifically looks for those files.

Compiling

```
csc.exe
/target:library
/r:$(DTFbin)\Microsoft.Deployment.WindowsInstaller.dll
/out:SampleCAs.dll
*.cs
```

Wrapping

MakeSfxCA.exe \$(OutDir)\SampleCAsPackage.dll \$(DTFbin)\SfxCA.dll SampleCAs.dll CustomAction.config \$(DTFbin)\Microsoft.Deployment.WindowsInstaller.dll Now the resulting package, SampleCAsPackage.dll, is ready to be inserted into the Binary table of the MSI.

For a working example of building a managed custom action package you can look at included sample ManagedCAs project.

See also:

- Writing Managed Custom Actions
- Specifying the Runtime Version

Deployment Tools Foundation Debugging Managed Custom Actions Development Guide > Managed CAs > Debugging

There are two ways to attach a debugger to a managed custom action.

Attach to message-box: Add some temporary code to your custom action to display a message box. Then when the message box pops up at install time, you can attch your debugger to that process (usually identifiable by the title of the message box). Once attached, you can ensure that symbols are loaded if necessary (they will be automatically loaded if PDB files were embedded in the CA assembly at build time), then set breakpoints anywhere in the custom action code.

MMsiBreak environment variable: When debugging *managed* custor actions, you should use the MMsiBreak environment variable instead of MsiBreak. Set the MMsiBreak variable to the custom action entrypoint name. (Remember this might be different from the method name if it wa overridden by the CustomActionAttribute.) When the CA proxy finds a matching name, the CLR JIT-debugging dialog will appear with text similar to "An exception 'Launch for user' has occurred in *YourCustomActionName*." The debug break occurs after the custom action assembly has been loaded, but just before custom action methoc is invoked. Once attached, you can ensure that symbols are loaded if necessary, then set breakpoints anywhere in the custom action code. Note: the MMsiBreak environment variable can also accept a commaseparated list of action names, any of which will cause a break when hit

Deployment Tools Foundation About InstallUtil Development Guide > Managed CAs > InstallUtil

InstallUtil is often considered as another option for executing MSI custo actions written in managed code. But in most cases it is not the best solution, for a number of reasons.

InstallUtil (in either InstallUtil.exe or InstallUtilLib.dll form) is a .NET Framework tool for executing the System.Configuration.Installer classes that are implemented in an assembly. That way the assembly can conta any special code required to install itself and uninstall itself. Essentially is the .NET replacement for COM self-registration aka DIIRegisterServe

Self-reg or System.Configuration.Installer is convenient for developmen use in order to test code without creating an actual setup package, or fc an IDE which wants to generate self-installing code. But experienced setup developers and the **Windows Installer documentation** all agree that self-reg is a bad practice for a production-quality setup. The current theory of state-of-the-art setup is that it should be as data-driven as possible. That is, the setup package describes as fully as possible the desired state of the system, and then the installer engine calculates the necessary actions to install, uninstall, patch, etc.

S.C.I encourages developers to write code for things such as registerin services or registering COM classes or other things which are more appropriately done using built-in MSI functionality (the ServiceInstall and Registry tables). The Visual Studio .NET wizards also tend to generate this kind of install code. Again, that is nice for development but not good for real installations. You end up with similar but slightly different code ir many places for doing the same thing. And that code is a black-box to tl installer engine.

An ideal MSI custom action is a logical extension of the setup engine, meaning it is data-driven and written in a very generic way to read from existing or custom tables in the MSI database, following a very similar pattern to the built-in actions. This makes the CA re-usable, and makes the installation more transparent. S.C.I custom actions invoked by InstallUtil cannot be data-driven because they don't have full access to the install session or database. They also cannot write to the install session's regular MSI log, but instead use a separate log which is bad for supportability.

InstallUtil also requires that the assembly be installed before the CA is able to execute. This is a problem for CAs that need to execute during the UI phase, or gather information before installation. For that purpose MSI allows custom action binaries to be embedded as non-installed file: but InstallUtil cannot make use of those.

Custom actions developed with DTF have none of the limitations of InstallUtil, giving a setup developer full capabilities to write well-designe custom actions, only now in managed code.

Deployment Tools Foundation Working with MSI Databases Development Guide > MSI Databases

Querying a database

```
using (Database db = new Database("product.msi", DatabaseOpenMode.Rea
{
    string value = (string) db.ExecuteScalar(
        "SELECT `Value` FROM `Property` WHERE `Property` = '{0}'", prop
}
```

1. Create a **new Database** instance referring to the location of the .msi or .msm file.

- 2. Execute the query:
 - The **ExecuteScalar** method is a shortcut for opening a view, executing the view, and fetching a single value.
 - The **ExecuteQuery** method is a shortcut for opening a view, executing the view, and fetching all values.
 - Or do it all manually with **Database.OpenView**, **View.Execute**, and **View.Fetch**.

Updating a binary

```
Database db = null;
View view = null;
Record rec = null;
try
{
    db = new Database("product.msi", DatabaseOpenMode.Direct);
    view = db.OpenView("UPDATE `Binary` SET `Data` = ? WHERE `Nam(
    rec = new Record(1);
    rec.SetStream(1, binFile);
    view.Execute(rec);
```

```
db.Commit();
}
finally
{
    if (rec != null) rec.Close();
    if (view != null) view.Close();
    if (db != null) db.Close();
}
```

1. Create a **new Database** instance referring to the location of the .msi or .msm file.

- 2. Open a view by calling one of the **Database.OpenView** overloads.
 - Parameters can be substituted in the SQL string using the String.Format syntax.
- 3. Create a record with one field containing the new binary value.
- 4. Execute the view by calling one of the View.Execute overloads.
 - A record can be supplied for substitution of field tokens (?) in the query.
- 5. Commit the Database.
- 6. Close the handles.

About handles

Handle objects (Database, View, Record, SummaryInfo, Session) will remain open until they are explicitly closed or until the objects are collected by the GC. So for the tightest code, handle objects should be explicitly closed when they are no longer needed, since closing them ca release significant resources, and too many unnecessary open handles can degrade performance. This is especially important within a loop construct: for example when iterating over all the Records in a table, it is much cleaner and faster to close each Record after it is used.

The handle classes in the managed library all extend the **InstallerHand** class, which implements the IDisposable interface. This makes them easily managed with C#'s using statement. Alternatively, they can be closed in a finally block.

As a general rule, *methods* in the library return new handle objects that should be managed and closed by the calling code, while *properties* onl return a reference to a prexisting handle object.

See also:

- MSI Diff Sample Tool
- Database Class

Send comments on this topic to wix-users@lists.sourceforge.net

Deployment Tools Foundation Working with Cabinet Files
Development Guide > Cabinet Files

Creating a cabinet

CabInfo cabInfo = new CabInfo("package.cab"); cabInfo.Pack("D:\\FilesToCompress");

1. Create a **new Cabinfo** instance referring to the (future) location of th .cab file.

- 2. Compress files:
 - Easily compress an entire directory with the **Pack** method.
 - Compress a specific list of exernal and internal filenames with the **PackFiles** method.
 - Compress a dictionary mapping of internal to external filenames with the PackFileSet method.

Listing a cabinet

CabInfo cabInfo = new CabInfo("package.cab"); foreach (CabFileInfo fileInfo in cabInfo.GetFiles()) Console.WriteLine(fileInfo.Name + "\t" + fileInfo.Length);

1. Create a **new CabInfo** instance referring to the location of the .cab file.

- 2. Enumerate files returned by the **GetFiles** method.
 - Each CabFileInfo instance contains metadata about one file.

Extracting a cabinet

CabInfo cabInfo = new CabInfo("package.cab"); cabInfo.Unpack("D:\\ExtractedFiles");

1. Create a **new CabInfo** instance referring to the location of the .cab file.

- 2. Extract files:
 - Easily extract all files to a directory with the Unpack method.
 - Easily extract a single file with the UnpackFile method.
 - Extract a specific list of filenames with the UnpackFiles method.
 - Extract a dictionary mapping of internal to external filenames with tl **UnpackFileSet** method.

Getting progress

Most cabinet operation methods have an overload that allows you to specify a event handler for receiving **archive progress events**. The **XPack sample** demonstrates use of the callback to report detailed progress to the console.

Stream-based compression

The CabEngine class contains static methods for performing compression/decompression operations directly on any kind of Stream. However these methods are more difficult to use, since the caller must implement a **stream context** that provides the file metadata which woul otherwise have been provided by the filesystem. The CabInfo class use the CabEngine class with FileStreams to provide the more traditional file based interface.

See also:

Cabinfo class

- CabEngine class
- XPack Sample Tool

Deployment Tools Foundation Working with Install Packages
Development Guide > Install Packages

Updating files in a product layout

The InstallPackage class makes it easy to work with files and cabinets i the context of a compressed or uncompressed product layout.

This hypothetical example takes an IDictionary 'files' which maps file ke to file paths. Each file is to be updated in the package layout; cabinets are even recompressed if necessary to include the new files.

```
using (InstallPackage pkg = new InstallPackage("d:\builds\product.msi",
DatabaseOpenMode.Transact))
{
    pkg.WorkingDirectory = Path.Combine(Path.GetTempFolder(), "pkgtmp"
    foreach (string fileKey in files.Keys)
    {
        string sourceFilePath = files[fileKey];
        string destFilePath = pkg.Files[fileKey].SourcePath;
        destFilePath = Path.Combine(pkg.WorkingDirectory, destFilePath);
        File.Copy(sourceFilePath, destFilePath, true);
    }
    pkg.UpdateFiles(new ArrayList(files.Keys));
    pkg.Commit();
    Directory.Delete(pkg.WorkingDirectory, true);
}
```

1. Create a **new InstallPackage** instance referring to the location of the .msi. This is actually just a specialized subclass of a Database.

2. Set the **WorkingDirectory**. This is the root directory where the package expects to find the new source files.

3. Copy each file to its proper location in the working directory. The **InstallPackage.Files** property is used to look up the relative source pat of each file.

4. Call InstallPackage.UpdateFiles with the list of file keys. This will re compress and package the files if necessary, and also update the following data: File.FileSize, File.Version, File.Language, MsiFileHash.HashPart*.

5. Commit the database changes and cleanup the working directory.

See also:

- WiFile Sample Tool a more complete tool that expands on the above example.
- InstallPackage Class

Deployment Tools Foundation Sample Applications Development Guide > Samples

Besides the simple managed custom action sample, there are three functional and useful sample tools included in this distribution:

MSI Inventory

Shows a hierarchical, relational, searchable view of all of the product, feature, component, file, and patch data managed by MSI, for all products installed on the system.

WiFile

Extracts and updates cabbed files in an MSI setup.

CabPack

Creates simple self-extracting cab packages. OK, so this one isn't especially useful as a tool, but the code should be helpful.

DDiff

Recursively diffs MSI, MSP, CAB, and other files and directories. Much more thorough than widiffdb.vbs.

Deployment Tools Foundation Windows Installer System Inventory Viewer

Development Guide > Samples > Inventory

This application shows a hierarchical, relational, searchable view of all (the product, feature, component, file, and patch data managed by MSI, for all products installed on the system.

Navigation

- 1. The tree on the left is self-explanatory.
- 2. Click on a row-header (grey box on the left side of the grid) to jump to a table with more details about the item referred to by that row. For example, clicking on a row-header of a table that lists components will take you to a table that lists the files in that component. Not every table has this ability, but the cursor will turn t a hand shape to indicate when this is possible.
- 3. Also you can navigate back and forward through your history using the buttons in the application or mouse buttons 4 and 5.

Searching

The search feature is not hard to find. By default, searches are limited to the current table. However, if you choose to find "In Subtree" by checkir the box, the search will include the current table as well as all tables under the current location in the tree. While this can take a long time if there is a lot of data under the current node, you can stop the search at any time with the stop button. The search pauses when a match is foun but clicking "Find" again will continue the same search from that point (unless you uncheck the "Continue" checkbox or change the search string).

Deployment Tools Foundation Windows Installer Package File Manipulation Tool

Development Guide > Samples > WiFile

Usage: WiFile.exe package.msi /l [filename,filename2,...] Usage: WiFile.exe package.msi /x [filename,filename2,...] Usage: WiFile.exe package.msi /u [filename,filename2,...]

Lists (/l), extracts (/x) or updates (/u) files in an MSI or MSM. Files are extracted using their source path relative to the package. Specified filenames do not include paths. Filenames may be a pattern such as *.exe or file?.dll

Example

The most powerful use of WiFile.exe is to do a round-trip update of files in a compressed MSI/MSM package. It works like this:

- Extract specific file(s) or all files from the package: WiFile.exe package.msi /x *
- 2. The files are now expanded into their directory structure. You can edit/update the files however you like.
- 3. Update the package with the changed files: WiFile.exe package.ms /u * This also updates the file metadata in the MSI including the fil version, size, and hash.

Notes

• Also works with packages that have multiple and/or external cab(s)

Deployment Tools Foundation Archive Pack/Unpack Tool Development Guide > Samples > XPack

Usage: CabPack.exe <directory> <package.cab> Usage: XPack /P <archive.cab> <directory> Usage: XPack /P <archive.zip> <directory>

Packs all files in a directory tree into an archive, using either the cab or zip format. Any existing archive with the same name will be overwritten.

Usage: XPack /U <archive.cab> <directory> Usage: XPack /U <archive.zip> <directory>

Unpacks all files from a cab or zip archive to the specified directory. Any existing files with the same names will be overwritten.

Deployment Tools Foundation MSI, MSP, CAB Diff Tool Development Guide > Samples > DDiff

MSI, MSP, CAB Diff Tool

Usage: DDiff target1 target2 [options] Example: DDiff d:\dir1 d:\dir2 Example: DDiff setup1.msi setup2.msi Example: DDiff patch1.msp patch2.msp -patchtarget target.msi Example: DDiff package1.cab package2.cab

Options:

- /o [filename] Output results to text file (UTF8)
- /p [package.msi] Diff patches relative to target MSI

The following types of inputs can be diffed:

- **Directories**: files and subdirectories are compared.
- **Cab files**: internal file list and files are compared.
- **MSI/MSM database files**: summary info, tables, and embedded binary and cab streams are compared.
- **MSP files**: summary info and embedded file cab are compared. When a patch target MSI is provided, the MSP's tables are also compared.
- Versioned files: Win32 file version is compared.
- Text files: if diff.exe is in the path, it is used to get a line-by-line diff
- Other files: file size and bytes are compared.

All processing is done recursively. So a versioned file within a cab within an MSI within a directory will have meaningful diff results.

Managed Libraries for Windows Installer

Proxy for Managed Custom Actions

The custom action proxy allows an MSI developer to write custom actions in managed code, while maintaing all the advantages of type 1 DLL custom actions including full access to installer state, properties, and the session database.

There are generally four problems that needed to be solved in order to create a type 1 custom action in managed code:

- 1. **Exporting the CA function as a native entry point callable by MSI:** The Windows Installer engine expects to call a LoadLibrary and GetProcAddress on the custom action DLL, so an unmanaged DLL needs to implement the function that is initially called by MSI and ultimately returns the result. This function acts as a proxy to relay the custom action call into the managed custom action assembly, and relay the result back to the caller.
- 2. **Providing supporting assemblies without requiring them to be installed as files:** If a DLL custom action runs before the product's files are installed, then it is difficult to provide any supporting files, because of the way the CA DLL is singly extracted and executed from a temp file. (This can be a problem for unmanaged CAs as well.) With managed custom actions we have already hit that problem since both the CA assembly and the MSI wrapper assembly need to be loaded. To solve this, the proxy DLL carries an appended cab package. When invoked, it will extract all contents of the cab package to a temporary working directory. This way the cab package can carry any arbitrary dependencies the custom action may require.

3. Hosting and configuring the Common Language Runtime: In

order to invoke a method in a managed assembly from a previously unmanaged process, the CLR needs to be "hosted". This involves choosing the correct version of the .NET Framework to use out of the available version(s) on the system, binding that version to the current process, and configuring it to load assemblies from the temporary working directory.

4. **Converting the integer session handle into a Session object:** The <u>Session</u> class in the managed wrapper library has a constructor which takes an integer session handle as its parameter. So the proxy simply instantiates this object before calling the real CA function.

The unmanaged CAPack module, when used in combination with the managed proxy in the Microsoft.WindowsInstaller assembly, accomplishes the tasks above to enable fully-functional managed DLL custom actions.

See also:

- <u>Writing Managed Custom Actions</u>
- Writing the CustomAction.config file
- <u>Sample C# Custom Actions</u>
- Building Managed Custom Actions