



Deployment Tools Foundation

**TIP**

[Deployment Tools Foundation reference documentation is available here.](#)

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.

Working with MSI Databases

Querying a database

```
using (Database db = new Database("product.msi", DatabaseOpenMode.ReadOnly))
{
    string value = (string) db.ExecuteScalar(
```

```
        "SELECT `Value` FROM `Property` WHERE `Property` = '{0}'", propName);  
    }
```

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 `Name` = '{0}'", binName))  
    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 can 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 InstallerHandle 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* only return a reference to a preexisting handle object.

Working with 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 the .cab file.
2. Compress files:
 - Easily compress an entire directory with the Pack method.
 - Compress a specific list of external 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 the 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 would otherwise have been provided by the filesystem. The CabInfo class uses the CabEngine class with FileStreams to provide the more traditional file-based interface.

Working with Install Packages

Updating files in a product layout

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

This hypothetical example takes an `IDictionary` `files` that maps file keys 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 path 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.

Managed Custom Actions

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. Your 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 of 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 -- use it.

Proxy for Managed Custom Actions

The custom action proxy allows an MSI developer to write custom actions in managed code, while maintaining 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 Session 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.

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 it 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 be managed and closed by the user code, while properties only return a reference to a preexisting 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.

Specifying the Runtime Version

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](#).

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 WixToolset.Dtf.WindowsInstaller.dll was built against.

CAUTION

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.

Sample C# Custom Action

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` = 'SampleCATest'");

    if(testProp == testProp2)
    {
        session.Log("Simple property test passed.");
        return ActionResult.Success;
    }
    else
        return ActionResult.Failure;
}
```

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

Compiling

```
csc.exe
    /target:library
```

```
/r:$(DTFbin)\WixToolset.Dtf.WindowsInstaller.dll  
/out:SampleCAs.dll  
*.cs
```

Wrapping

```
MakeSfxCA.exe  
$(OutDir)\SampleCAsPackage.dll  
$(DTFbin)\SfxCA.dll  
SampleCAs.dll  
CustomAction.config  
$(DTFbin)\WixToolset.Dtf.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.

Debugging Managed Custom Actions

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 attach 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 custom 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 was 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 method 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 comma-separated list of action names, any of which will cause a break when hit.

 [Edit this page](#)