**Types of Assemblies**

.NET Framework 4

**Assemblies in the Common Language Runtime**

Assemblies are the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An assembly provides the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly.

An assembly performs the following functions:

* It contains code that the common language runtime executes. Microsoft intermediate language (MSIL) code in a portable executable (PE) file will not be executed if it does not have an associated assembly manifest. Note that each assembly can have only one entry point (that is, **DllMain**, **WinMain**, or **Main**).
* It forms a security boundary. An assembly is the unit at which permissions are requested and granted. For more information about security boundaries as they apply to assemblies, see [Assembly Security Considerations](http://msdn.microsoft.com/en-us/library/ab4eace3.aspx).
* It forms a type boundary. Every type's identity includes the name of the assembly in which it resides. A type called MyType that is loaded in the scope of one assembly is not the same as a type called MyType that is loaded in the scope of another assembly.
* It forms a reference scope boundary. The assembly's manifest contains assembly metadata that is used for resolving types and satisfying resource requests. It specifies the types and resources that are exposed outside the assembly. The manifest also enumerates other assemblies on which it depends.
* It forms a version boundary. The assembly is the smallest versionable unit in the common language runtime; all types and resources in the same assembly are versioned as a unit. The assembly's manifest describes the version dependencies you specify for any dependent assemblies. For more information about versioning, see [Assembly Versioning](http://msdn.microsoft.com/en-us/library/51ket42z.aspx).
* It forms a deployment unit. When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as localization resources or assemblies containing utility classes, can be retrieved on demand. This allows applications to be kept simple and thin when first downloaded. For more information about deploying assemblies, see [Deploying Applications](http://msdn.microsoft.com/en-us/library/6hbb4k3e.aspx).
* It is the unit at which side-by-side execution is supported. For more information about running multiple versions of an assembly, see [Assemblies and Side-by-Side Execution](http://msdn.microsoft.com/en-us/library/fdhkd3a5.aspx).

Assemblies can be static or dynamic. Static assemblies can include .NET Framework types (interfaces and classes), as well as resources for the assembly (bitmaps, JPEG files, resource files, and so on). Static assemblies are stored on disk in portable executable (PE) files. You can also use the .NET Framework to create dynamic assemblies, which are run directly from memory and are not saved to disk before execution. You can save dynamic assemblies to disk after they have executed.

There are several ways to create assemblies. You can use development tools, such as Visual Studio 2005, that you have used in the past to create .dll or .exe files. You can use tools provided in the Windows Software Development Kit (SDK) to create assemblies with modules created in other development environments. You can also use common language runtime APIs, such as [Reflection.Emit](http://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx), to create dynamic assemblies.

.Net Assembly can be classified in four Categories:

1. **With Respect to Program Access**

* **Private Assembly:** It can be used only in one application. It is the default style of assembly. And must reside in the application folder.
* **Public/Shared Assembly:** It can be used by all applications in the server. It must have a globally unique name and must be installed in the GAC (Global Assembly Cache).

A private assembly is used only by a single application, and is stored in that application's install directory (or a subdirectory therein). A shared assembly is one that can be referenced by more than one application. In order to share an assembly, the assembly must be explicitly built for this purpose by giving it a cryptographically strong name (referred to as a strong name). By contrast, a private assembly name need only be unique within the application that uses it.

By making a distinction between private and shared assemblies, we introduce the notion of sharing as an explicit decision. Simply by deploying private assemblies to an application directory, you can guarantee that that application will run only with the bits it was built and deployed with. References to private assemblies will only be resolved locally to the private application directory.

There are several reasons you may elect to build and use shared assemblies, such as the ability to express version policy. The fact that shared assemblies have a cryptographically strong name means that only the author of the assembly has the key to produce a new version of that assembly. Thus, if you make a policy statement that says you want to accept a new version of an assembly, you can have some confidence that version updates will be controlled and verified by the author. Otherwise, you don't have to accept them.

For locally installed applications, a shared assembly is typically explicitly installed into the global assembly cache (a local cache of assemblies maintained by the .NET Framework). Key to the version management features of the .NET Framework is that downloaded code does not affect the execution of locally installed applications. Downloaded code is put in a special download cache and is not globally available on the machine even if some of the downloaded components are built as shared assemblies.

The classes that ship with the .NET Framework are all built as shared assemblies.

1. **With Respect to Number of Resources**

* **Static Assembly:** Assemblies can be static or dynamic. Static assemblies can include .NET Framework types (interfaces and classes), as well as resources for the assembly (bitmaps, JPEG files, resource files, and so on). Static assemblies are stored on disk in portable executable (PE) files. You can also use the .NET Framework to create dynamic assemblies, which are run directly from memory and are not saved to disk before execution. You can save dynamic assemblies to disk after they have executed.
* **Dynamic Assembly:** Reflection emit provides many ways to create dynamic assemblies. Dynamic assemblies can be created using the various System.AppDomain.DefineDynamicAssembly methods. DefineDynamicAssembly returns an AssemblyBuilder object. DefineDynamicAssembly requires the caller to specify the AssemblyBuilderAccess enumeration value. The enumeration value specifies whether the dynamic assembly will be run only, saved only, or run and/or saved. Some of the methods require the caller to supply evidence, which is the set of information that constitutes input to security policy decisions, such as which permissions can be granted to code. Other methods require the caller to request permissions. Three kinds of permission requests exist: required, optional, and refused.

1. **With Respect to Deployment**

* **Satellite Assembly:** Use the Assembly Linker (Al.exe) to compile .resources files into satellite assemblies. Al.exe creates an assembly from the .resources files that you specify. By definition, satellite assemblies can only contain resources. They cannot contain any executable code.
* **Resource-Only Assembly:** Resource assemblies are useful when you need to frequently update resources in a program without having to recompile the entire solution. By placing all of your resources into a resource-only assembly, you can update the assembly as needed without having to recompile your application. Resources compiled into resource assemblies can be retrieved using the ResourceManager class.

1. **With Respect to Number of Assemblies**

* **Single-file Assembly**
* **Multifile Assembly**

For an assembly with one associated file, the manifest is incorporated into the PE file to form a single-file assembly. You can create a multifile assembly with a standalone manifest file or with the manifest incorporated into one of the PE files in the assembly (vide pix in the next page).

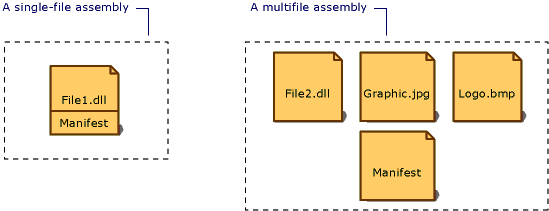
.NET Framework 4

**Assembly Manifest**

Every assembly, whether static or dynamic, contains a collection of data that describes how the elements in the assembly relate to each other. The assembly manifest contains this assembly metadata. An assembly manifest contains all the metadata needed to specify the assembly's version requirements and security identity, and all metadata needed to define the scope of the assembly and resolve references to resources and classes. The assembly manifest can be stored in either a PE file (an .exe or .dll) with Microsoft intermediate language (MSIL) code or in a standalone PE file that contains only assembly manifest information.

The following illustration shows the different ways the manifest can be stored.

**Types of assemblies**



For an assembly with one associated file, the manifest is incorporated into the PE file to form a single-file assembly. You can create a multifile assembly with a standalone manifest file or with the manifest incorporated into one of the PE files in the assembly.

Each assembly's manifest performs the following functions:

* Enumerates the files that make up the assembly.
* Governs how references to the assembly's types and resources map to the files that contain their declarations and implementations.
* Enumerates other assemblies on which the assembly depends.
* Provides a level of indirection between consumers of the assembly and the assembly's implementation details.
* Renders the assembly self-describing.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifAssembly Manifest Contents

The following table shows the information contained in the assembly manifest. The first four items—the assembly name, version number, culture, and strong name information—make up the assembly's identity.

|  |  |
| --- | --- |
| **Information** | **Description** |
| Assembly name | A text string specifying the assembly's name. |
| Version number | A major and minor version number, and a revision and build number. The common language runtime uses these numbers to enforce version policy. |
| Culture | Information on the culture or language the assembly supports. This information should be used only to designate an assembly as a satellite assembly containing culture- or language-specific information. (An assembly with culture information is automatically assumed to be a satellite assembly.) |
| Strong name information | The public key from the publisher if the assembly has been given a strong name. |
| List of all files in the assembly | A hash of each file contained in the assembly and a file name. Note that all files that make up the assembly must be in the same directory as the file containing the assembly manifest. |
| Type reference information | Information used by the runtime to map a type reference to the file that contains its declaration and implementation. This is used for types that are exported from the assembly. |
| Information on referenced assemblies | A list of other assemblies that are statically referenced by the assembly. Each reference includes the dependent assembly's name, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strong named. |

You can add or change some information in the assembly manifest by using assembly attributes in your code. You can change version information and informational attributes, including Trademark, Copyright, Product, Company, and Informational Version. For a complete list of assembly attributes, see [Setting Assembly Attributes](http://msdn.microsoft.com/en-us/library/4w8c1y2s.aspx).

.NET Framework 4

**How to: Install an Assembly into the Global Assembly Cache**

Updated: June 2010

There are three ways to install an assembly into the global assembly cache:

* Using the [Global Assembly Cache tool (Gacutil.exe)](http://msdn.microsoft.com/en-us/library/ex0ss12c.aspx).

You can use Gacutil.exe to add strong-named assemblies to the global assembly cache and to view the contents of the global assembly cache.

|  |
| --- |
| **Description: NoteNote** |
| Gacutil.exe is only for development purposes and should not be used to install production assemblies into the global assembly cache. |

* Using Microsoft Windows Installer 2.0.

This is the recommended and most common way to add assemblies to the global assembly cache. The installer provides reference counting of assemblies in the global assembly cache, plus other benefits.

* Using the [Mscorcfg.msc (.NET Framework Configuration Tool)](http://msdn.microsoft.com/en-us/library/2bc0cxhc.aspx).

The [Mscorcfg.msc (.NET Framework Configuration Tool)](http://msdn.microsoft.com/en-us/library/2bc0cxhc.aspx) allows you to view the global assembly cache and add new assemblies to the cache.

|  |
| --- |
| **Description: NoteNote** |
| In earlier versions of the .NET Framework, the [Shfusion.dll](http://msdn.microsoft.com/en-us/library/34149zk3.aspx) Windows shell extension enabled you to install assemblies by dragging them in Windows Explorer. Beginning with the .NET Framework version 4, Shfusion.dll is obsolete. |

### To install a strong-named assembly into the global assembly cache using the Global Assembly Cache tool (Gacutil.exe)

* At the command prompt, type the following command:

**gacutil –I** <assembly name>

In this command, assembly name is the name of the assembly to install in the global assembly cache.

The following example installs an assembly with the file name hello.dll into the global assembly cache.

gacutil -i hello.dll

|  |
| --- |
| **Important noteImportant** |
| Mscorcfg.msc has been removed from the .NET Framework version 4 and later versions. This documentation applies only to earlier versions of the .NET Framework. For more information about security changes in the .NET Framework 4, see [Security Changes in the .NET Framework 4](http://msdn.microsoft.com/en-us/library/dd233103.aspx). |

.NET Framework 4

**Mscorcfg.msc (.NET Framework Configuration Tool)**

The .NET Framework Configuration tool (Mscorcfg.msc) is a Microsoft Management Console (MMC) snap-in that enables you to manage and configure assemblies in the [global assembly cache](http://msdn.microsoft.com/en-us/library/6axd4fx6.aspx) and adjust code access security policy.

|  |
| --- |
| **Important noteImportant** |
| Mscorcfg.msc has been removed from the .NET Framework version 4 and later versions. This documentation applies only to earlier versions of the .NET Framework. For more information about security changes in the .NET Framework 4, see [Security Changes in the .NET Framework 4](http://msdn.microsoft.com/en-us/library/dd233103.aspx). |

In the .NET Framework versions 1.0 and 1.1, Mscorcfg.msc is installed with the NET Framework redistributable package. In the .NET Framework 2.0 and later versions, Mscorcfg.msc is installed with the [.NET Framework 2.0 Software Development Kit (SDK)](http://go.microsoft.com/fwlink/?LinkId=115253).

If you have both the .NET Framework 1.1 and 2.0 runtimes, you will have version 1.1 of the configuration tool, but you might not have version 2.0. If you want to manage .NET Framework 2.0, 3.0, or 3.5 by using the configuration tool, you must install the .NET Framework 2.0 SDK. To change configuration settings for a computer that has multiple versions of the .NET Framework, you must make the changes in the matching versions of the configuration tool.

|  |
| --- |
| **NoteNote** |
| The .NET Framework versions 3.0 and 3.5 are built incrementally on the .NET Framework version 2.0. The configuration tool included in the .NET Framework 2.0 SDK is the latest version of the tool. You can use this version to manage code access security policy for the .NET Framework 3.0 and 3.5. |