Assemblies in .Net

When we create a project in Visual Studio .Net and compile it, assemblies are created. These **assemblies** are the fundamental units of applications in the .NET Framework. An assembly can contain classes, structures, interfaces, and resources that an application requires.

When we compile a program, the compiler translates the source code into the **Intermediate Language code (IL)**. In addition to translating the code into IL, the compiler also produces **metadata** about the program during the process of the compilation. Metadata contains the description of the program, such as the classes and interfaces, the dependencies, and the versions of the components used in the program.

The IL and the metadata are linked in an assembly.

An IL and metadata exist in **portable executable file**(EXE/DLL).

If we look at any project folder, we will find a bin folder that contains, say myproject.dll/ myproject.exe and myproject.pdb files. The myproject.dll/ myproject.exe is the assembly, and myproject.pdb (program database) file contains debugging information for the assembly. The .pdb file also contains information the linker can use when debugging the assembly.

So, when we create and compile the project in VS.Net, the assembly is automatically generated and copied into the application's bin directory. An assembly also contains an **assembly manifest** that contains the assembly metadata. This metadata contains information about the assembly version, its security identity, the resources required by the assembly and the scope of the assembly. This information is stored within the assembly file (DLL/EXE) itself. Note that the assembly contains two types of metadata. One, is the type metadata and the other is the assembly metadata.

Process assemblies (EXE) and library assemblies(DLL)

A process assembly represents a process which uses classes defined in library assemblies. The compiler will have a switch to determine if the assembly is a process or a library and will set a flag in the PE file. .NET does not use the extension to determine if the file is a process or library. This means that a library may have either .dll or .exe as its extension.

The .Net CLR while executing looks for the metadata in order to locate and load classes, as well as generate native code and security.

Note that the assembly metadata is assembly manifest and it simplifies the deployment model. Because it contains so much information about the assembly, you can simply XCOPY the assembly onto another computer, along with its related manifest.

Why use Assemblies?

The goal of the assembly model is the elimination of DLL Hell. Under the current COM/COM+ model, a catalog of DLLs is centralized in the Windows Registry. When a new version of a DLL is published, the registry re-references the catalog to point to the new DLL. This centralized registration paradigm makes it challenging for multiple applications to depend on the same DLL.

Most often, the application binds to a DLL in a centralized location, rather than run multiple versions of a component by using side-by-side execution.

The .NET Framework makes it easy to run multiple versions of a component because it stores assemblies in local application directories by default. This isolates the assembly from use by any other application and protects the assembly from system changes.

Types of Assemblies

Assemblies can be **private or shared**. The assembly which is used by a single application is called as **private assembly**. Suppose we have created a DLL which encapsulates business logic. This DLL is used by the client application only and not by any other application. In order to run the application properly the DLL must reside in the same folder in which the client application is installed. Thus the assembly is private to the application.

Suppose we are creating a general purpose DLL which provides functionality to be used by a variety of applications. Now, instead of each client application having its own copy of DLL we can place the DLL in 'global assembly cache'. Such assemblies are called as **shared assemblies**.

When we create an application in Visual Studio.Net that uses a component, which is not in the GAC, the assembly is automatically copied to the project´s bin folder. So, if we simply the copy the project´s bin folder, the assembly for the component is copied as well. Since the files the application needs are in the same folder, the application can run without registration.

Benefits of Private Assemblies

Private assemblies are installed in a directory named bin located under the application directory. These files are private to the application.

No versioning is required, as long as it is the same version as the one with which the application was built.
It enables XCOPY deployment
There is no configuration or signing
It is great for small utility Assemblies/ application specific code.

Disadvantages:

When you have multiple applications using one assembly, you have to deploy the assembly to the bin directory of each application.

How to create an Assembly

A shared assembly can be created in Visual Studio.Net by generating Cryptographic Key Pair using the tool SN.exe utility. The sn.exe utility is located in the Bin directory. After creating the shared assembly we should sign the shared assembly to the GAC by using Gacutil tool.

What is Global Assembly Cache?

An assembly that is shared by multiple applications is called a global assembly and is installed in the **global assembly cache (GAC)**.

Global assembly cache is nothing but a special disk folder where all the shared assemblies are kept. It is located under <drive>:\Windows\Assembly folder on every machine running the .NET framework.

GAC acts as a central location for assemblies that is accessible by any application.

Summary:

1. Assembly physically exist as DLLs or EXEs.
2. An assembly consists of a manifest and the portable executables (PE).
3. Global assembly cache is nothing but a special disk folder where all the shared assemblies will be kept.