Assemblies are the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for .NET-based applications. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. Assemblies take the form of executable (*.exe*) or dynamic link library (*.dll*) files, and are the building blocks of .NET applications. They provide the common language runtime with the information it needs to be aware of type implementations.

In .NET and .NET Framework, you can build an assembly from one or more source code files. In .NET Framework, assemblies can contain one or more modules. This way, larger projects can be planned so that several developers can work on separate source code files or modules, which are combined to create a single assembly. For more information about modules, see How to: Build a multifile assembly.

Assemblies have the following properties:

- Assemblies are implemented as *.exe* or *.dll* files.
- For libraries that target .NET Framework, you can share assemblies between applications by putting them in the global assembly cache (GAC). You must strong-name assemblies before you can include them in the GAC. For more information, see Strong-named assemblies.
- Assemblies are only loaded into memory if they're required. If they aren't used, they aren't loaded. Therefore, assemblies can be an efficient way to manage resources in larger projects.
- You can programmatically obtain information about an assembly by using reflection. For more information, see Reflection (C#) or Reflection (Visual Basic).
- You can load an assembly just to inspect it by using the MetadataLoadContext class on .NET and .NET Framework. MetadataLoadContext replaces the Assembly.ReflectionOnlyLoad methods.

Assemblies in the common language runtime

Assemblies provide the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type doesn't exist outside the context of an assembly.

An assembly defines the following information:

- **Code** that the common language runtime executes. Each assembly can have only one entry point: `DllMain`, `WinMain`, or `Main`.
- The **security boundary**. An assembly is the unit at which permissions are requested and granted. For more information about security boundaries in assemblies, see Assembly security considerations.
- The **type boundary**. Every type's identity includes the name of the assembly in which it resides. A type called `MyType` that's loaded in the scope of one assembly isn't the same as a type called `MyType` that's loaded in the scope of another assembly.
- The **reference-scope boundary**: The assembly manifest has metadata that's used for resolving types and satisfying resource requests. The manifest specifies the types and resources to expose outside the assembly and enumerates other assemblies on which it depends. Microsoft intermediate language (MSIL) code in a portable executable (PE) file won't be executed unless it has an associated assembly manifest.
- The **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 manifest describes the version dependencies you specify for any dependent assemblies. For more information about versioning, see Assembly versioning.
- The **deployment unit**: When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as assemblies containing localization resources or utility classes, can be retrieved on demand. This process allows apps to be simple and thin when first downloaded. For more information about deploying assemblies, see Deploy applications.
- A **side-by-side execution unit**: For more information about running multiple versions of an assembly, see Assemblies and side-by-side execution.

Create an assembly

Assemblies can be static or dynamic. Static assemblies are stored on a disk in portable executable (PE) files. Static assemblies can include interfaces, classes, and resources like bitmaps, JPEG files, and other resource files. You can also create dynamic assemblies, which

are run directly from memory and aren't saved to disk before execution. You can save dynamic assemblies to disk after they've been executed.

There are several ways to create assemblies. You can use development tools, such as Visual Studio that can create *.dll* or *.exe* files. You can use tools in the Windows SDK to create assemblies with modules from other development environments. You can also use common language runtime APIs, such as System.Reflection.Emit, to create dynamic assemblies.

Compile assemblies by building them in Visual Studio, building them with .NET Core command-line interface tools, or building .NET Framework assemblies with a command-line compiler. For more information about building assemblies using .NET CLI, see .NET CLI overview.

> **Note**
>
> To build an assembly in Visual Studio, on the **Build** menu, select **Build**.

Assembly manifest

Every assembly has an *assembly manifest* file. Similar to a table of contents, the assembly manifest contains:

- The assembly's identity (its name and version).
- A file table describing all the other files that make up the assembly, such as other assemblies you created that your *.exe* or *.dll* file relies on, bitmap files, or Readme files.
- An *assembly reference list*, which is a list of all external dependencies, such as *.dll*s or other files. Assembly references contain references to both global and private objects. Global objects are available to all other applications. In .NET Core, global objects are coupled with a particular .NET Core runtime. In .NET Framework, global objects reside in the global assembly cache (GAC). *System.IO.dll* is an example of an assembly in the GAC. Private objects must be in a directory level at or below the directory in which your app is installed.

Assemblies contain information about content, versioning, and dependencies. So the applications that use them don't need to rely on external sources, such as the registry on Windows systems, to function properly. Assemblies reduce *.dll* conflicts and make your

applications more reliable and easier to deploy. In many cases, you can install a .NET-based application simply by copying its files to the target computer. For more information, see Assembly manifest.

Add a reference to an assembly

To use an assembly in an application, you must add a reference to it. When an assembly is referenced, all the accessible types, properties, methods, and other members of its namespaces are available to your application as if their code were part of your source file.

> **Note**
>
> Most assemblies from the .NET Class Library are referenced automatically. If a system assembly isn't automatically referenced, add a reference in one of the following ways:
>
> - For .NET and .NET Core, add a reference to the NuGet package that contains the assembly. Either use the NuGet Package Manager in Visual Studio or add a **<PackageReference>** element for the assembly to the *.csproj* or *.vbproj* project.
> - For .NET Framework, add a reference to the assembly by using the **Add Reference** dialog in Visual Studio or the `-reference` command line option for the **C#** or **Visual Basic** compilers.

In C#, you can use two versions of the same assembly in a single application. For more information, see extern alias.

Related content

Title	Description
Assembly contents	Elements that make up an assembly.
Assembly manifest	Data in the assembly manifest, and how it's stored in assemblies.
Global assembly cache	How the GAC stores and uses assemblies.

Title	Description
Strong-named assemblies	Characteristics of strong-named assemblies.

Assembly security considerations	How security works with assemblies.
Assembly versioning	Overview of the .NET Framework versioning policy.
Assembly placement	Where to locate assemblies.
Assemblies and side-by-side execution	Use multiple versions of the runtime or an assembly simultaneously.
Emit dynamic methods and assemblies	How to create dynamic assemblies.
How the runtime locates assemblies	How the .NET Framework resolves assembly references at run time.

Reference

System.Reflection.Assembly

See also

- .NET assembly file format
- Friend assemblies
- Reference assemblies
- How to: Load and unload assemblies
- How to: Use and debug assembly unloadability in .NET Core
- How to: Determine if a file is an assembly
- How to: Inspect assembly contents using MetadataLoadContext