Common Language Runtime (CLR) overview

Article • 04/25/2023

.NET provides a run-time environment called the common language runtime that runs the code and provides services that make the development process easier.

Compilers and tools expose the common language runtime's functionality and enable you to write code that benefits from the managed execution environment. Code that you develop with a language compiler that targets the runtime is called managed code. Managed code benefits from features such as cross-language integration, cross-language exception handling, enhanced security, versioning and deployment support, a simplified model for component interaction, and debugging and profiling services.

> **Note**
>
> Compilers and tools can produce output that the common language runtime can consume because the type system, the format of metadata, and the run-time environment (the virtual execution system) are all defined by a public standard, the ECMA Common Language Infrastructure specification. For more information, see **ECMA C# and Common Language Infrastructure Specifications**.

To enable the runtime to provide services to managed code, language compilers must emit metadata that describes the types, members, and references in your code. Metadata is stored with the code; every loadable common language runtime portable executable (PE) file contains metadata. The runtime uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run-time context boundaries.

The runtime automatically handles object layout and manages references to objects, releasing them when they're no longer being used. Objects whose lifetimes are managed in this way are called managed data. Garbage collection eliminates memory leaks and some other common programming errors. If your code is managed, you can use managed, unmanaged, or both managed and unmanaged data in your .NET application. Because language compilers supply their own types, such as primitive types, you might not always know or need to know whether your data is being managed.

The common language runtime makes it easy to design components and applications whose objects interact across languages. Objects written in different languages can communicate with each other, and their behaviors can be tightly integrated. For

example, you can define a class and then use a different language to derive a class from your original class or call a method on the original class. You can also pass an instance of a class to a method of a class written in a different language. This cross-language integration is possible because language compilers and tools that target the runtime use a common type system defined by the runtime. They follow the runtime's rules for defining new types and for creating, using, persisting, and binding to types.

As part of their metadata, all managed components carry information about the components and resources they were built against. The runtime uses this information to ensure that your component or application has the specified versions of everything it needs, which makes your code less likely to break because of some unmet dependency. Registration information and state data are no longer stored in the registry, where they can be difficult to establish and maintain. Instead, information about the types you define and their dependencies is stored with the code as metadata. This way, the task of component replication and removal is less complicated.

Language compilers and tools expose the runtime's functionality in ways that are intended to be useful and intuitive to developers. Some features of the runtime might be more noticeable in one environment than in another. How you experience the runtime depends on which language compilers or tools you use. For example, if you're a Visual Basic developer, you might notice that with the common language runtime, the Visual Basic language has more object-oriented features than before. The runtime provides the following benefits:

- Performance improvements.
- The ability to easily use components developed in other languages.
- Extensible types provided by a class library.
- Language features such as inheritance, interfaces, and overloading for object-oriented programming.
- Support for explicit free threading that allows creation of multithreaded and scalable applications.
- Support for structured exception handling.
- Support for custom attributes.
- Garbage collection.
- Use of delegates instead of function pointers for increased type safety and security. For more information about delegates, see Common Type System.

CLR versions

.NET Core and .NET 5+ releases have a single product version, that is, there's no separate CLR version. For a list of .NET Core versions, see Download .NET Core.

However, the .NET Framework version number doesn't necessarily correspond to the version number of the CLR it includes. For a list of .NET Framework versions and their corresponding CLR versions, see .NET Framework versions and dependencies.

Related articles

Title	Description
Managed Execution Process	Describes the steps required to take advantage of the common language runtime.
Automatic Memory Management	Describes how the garbage collector allocates and releases memory.
Overview of .NET Framework	Describes key .NET Framework concepts, such as the common type system, cross-language interoperability, managed execution, application domains, and assemblies.
Common Type System	Describes how types are declared, used, and managed in the runtime in support of cross-language integration.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Managed Execution Process

Article • 09/15/2021

The managed execution process includes the following steps, which are discussed in detail later in this topic:

1. Choosing a compiler.

   To obtain the benefits provided by the common language runtime, you must use one or more language compilers that target the runtime.

2. Compiling your code to MSIL.

   Compiling translates your source code into Microsoft intermediate language (MSIL) and generates the required metadata.

3. Compiling MSIL to native code.

   At execution time, a just-in-time (JIT) compiler translates the MSIL into native code. During this compilation, code must pass a verification process that examines the MSIL and metadata to find out whether the code can be determined to be type safe.

4. Running code.

   The common language runtime provides the infrastructure that enables execution to take place and services that can be used during execution.

Choosing a Compiler

To obtain the benefits provided by the common language runtime (CLR), you must use one or more language compilers that target the runtime, such as Visual Basic, C#, Visual C++, F#, or one of many third-party compilers such as an Eiffel, Perl, or COBOL compiler.

Because it is a multilanguage execution environment, the runtime supports a wide variety of data types and language features. The language compiler you use determines which runtime features are available, and you design your code using those features. Your compiler, not the runtime, establishes the syntax your code must use. If your component must be completely usable by components written in other languages, your component's exported types must expose only language features that are included in the Common Language Specification (CLS). You can use the CLSCompliantAttribute attribute to ensure that your code is CLS-compliant. For more information, see Language independence and language-independent components.

Back to top

Compiling to MSIL

When compiling to managed code, the compiler translates your source code into Microsoft intermediate language (MSIL), which is a CPU-independent set of instructions that can be efficiently converted to native code. MSIL includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other operations. Before code can be run, MSIL must be converted to CPU-specific code, usually by a just-in-time (JIT) compiler. Because the common language runtime supplies one or more JIT compilers for each computer architecture it supports, the same set of MSIL can be JIT-compiled and run on any supported architecture.

When a compiler produces MSIL, it also produces metadata. Metadata describes the types in your code, including the definition of each type, the signatures of each type's members, the members that your code references, and other data that the runtime uses at execution time. The MSIL and metadata are contained in a portable executable (PE) file that is based on and that extends the published Microsoft PE and common object file format (COFF) used historically for executable content. This file format, which accommodates MSIL or native code as well as metadata, enables the operating system to recognize common language runtime images. The presence of metadata in the file together with MSIL enables your code to describe itself, which means that there is no need for type libraries or Interface Definition Language (IDL). The runtime locates and extracts the metadata from the file as needed during execution.

Back to top

Compiling MSIL to Native Code

Before you can run Microsoft intermediate language (MSIL), it must be compiled against the common language runtime to native code for the target machine architecture. .NET provides two ways to perform this conversion:

- A .NET just-in-time (JIT) compiler.
- Ngen.exe (Native Image Generator).

Compilation by the JIT Compiler

JIT compilation converts MSIL to native code on demand at application run time, when the contents of an assembly are loaded and executed. Because the common language runtime supplies a JIT compiler for each supported CPU architecture, developers can build a set of MSIL assemblies that can be JIT-compiled and run on different computers with different machine architectures. However, if your managed code calls platform-specific native APIs or a platform-specific class library, it will run only on that operating system.

JIT compilation takes into account the possibility that some code might never be called during execution. Instead of using time and memory to convert all the MSIL in a PE file to native code, it converts the MSIL as needed during execution and stores the resulting native code in memory so that it is accessible for subsequent calls in the context of that process. The loader creates and attaches a stub to each method in a type when the type is loaded and initialized. When a method is called for the first time, the stub passes control to the JIT compiler, which converts the MSIL for that method into native code and modifies the stub to point directly to the generated native code. Therefore, subsequent calls to the JIT-compiled method go directly to the native code.

Install-Time Code Generation Using NGen.exe

Because the JIT compiler converts an assembly's MSIL to native code when individual methods defined in that assembly are called, it affects performance adversely at run time. In most cases, that diminished performance is acceptable. More importantly, the code generated by the JIT compiler is bound to the process that triggered the compilation. It cannot be shared across multiple processes. To allow the generated code to be shared across multiple invocations of an application or across multiple processes that share a set of assemblies, the common language runtime supports an ahead-of-time compilation mode. This ahead-of-time compilation mode uses the Ngen.exe (Native Image Generator) to convert MSIL assemblies to native code much like the JIT compiler does. However, the operation of Ngen.exe differs from that of the JIT compiler in three ways:

- It performs the conversion from MSIL to native code before running the application instead of while the application is running.
- It compiles an entire assembly at a time, instead of one method at a time.
- It persists the generated code in the Native Image Cache as a file on disk.

Code Verification

As part of its compilation to native code, the MSIL code must pass a verification process unless an administrator has established a security policy that allows the code to bypass verification. Verification examines MSIL and metadata to find out whether the code is type safe, which means that it accesses only the memory locations it is authorized to access. Type safety helps isolate objects from each other and helps protect them from inadvertent or malicious corruption. It also provides assurance that security restrictions on code can be reliably enforced.

The runtime relies on the fact that the following statements are true for code that is verifiably type safe:

- A reference to a type is strictly compatible with the type being referenced.
- Only appropriately defined operations are invoked on an object.
- Identities are what they claim to be.

During the verification process, MSIL code is examined in an attempt to confirm that the code can access memory locations and call methods only through properly defined types. For example, code cannot allow an object's fields to be accessed in a manner that allows memory locations to be overrun. Additionally, verification inspects code to determine whether the MSIL has been correctly generated, because incorrect MSIL can lead to a violation of the type safety rules. The verification process passes a well-defined set of type-safe code, and it passes only code that is type safe. However, some type-safe code might not pass verification because of some limitations of the verification process, and some languages, by design, do not produce verifiably type-safe code. If type-safe code is required by the security policy but the code does not pass verification, an exception is thrown when the code is run.

Back to top

Running Code

The common language runtime provides the infrastructure that enables managed execution to take place and services that can be used during execution. Before a method can be run, it must be compiled to processor-specific code. Each method for which MSIL has been generated is JIT-compiled when it is called for the first time, and then run. The next time the method is run, the existing JIT-compiled native code is run. The process of JIT-compiling and then running the code is repeated until execution is complete.

During execution, managed code receives services such as garbage collection, security, interoperability with unmanaged code, cross-language debugging support, and

enhanced deployment and versioning support.

In Microsoft Windows Vista, the operating system loader checks for managed modules by examining a bit in the COFF header. The bit being set denotes a managed module. If the loader detects managed modules, it loads mscoree.dll, and `_CorValidateImage` and `_CorImageUnloading` notify the loader when the managed module images are loaded and unloaded. `_CorValidateImage` performs the following actions:

1. Ensures that the code is valid managed code.
2. Changes the entry point in the image to an entry point in the runtime.

On 64-bit Windows, `_CorValidateImage` modifies the image that is in memory by transforming it from PE32 to PE32+ format.

Back to top

See also

- Overview
- Language independence and language-independent components
- Metadata and Self-Describing Components
- Ilasm.exe (IL Assembler)
- Security
- Interoperating with Unmanaged Code
- Deployment
- Assemblies in .NET
- Application Domains

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Assemblies in .NET

Article • 03/15/2023

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.
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.

Title	Description
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

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Assembly contents

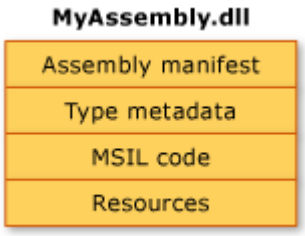
Article • 09/15/2021

In general, a static assembly can consist of four elements:

- The assembly manifest, which contains assembly metadata.
- Type metadata.
- Microsoft intermediate language (MSIL) code that implements the types. It is generated by the compiler from one or more source code files.
- A set of resources.

Only the assembly manifest is required, but either types or resources are needed to give the assembly any meaningful functionality.

The following illustration shows these elements grouped into a single physical file:



As you design your source code, you make explicit decisions about how to partition the functionality of your application into one or more files. When designing .NET code, you'll make similar decisions about how to partition the functionality into one or more assemblies.

See also

- Assemblies in .NET
- Assembly manifest
- Assembly security considerations

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

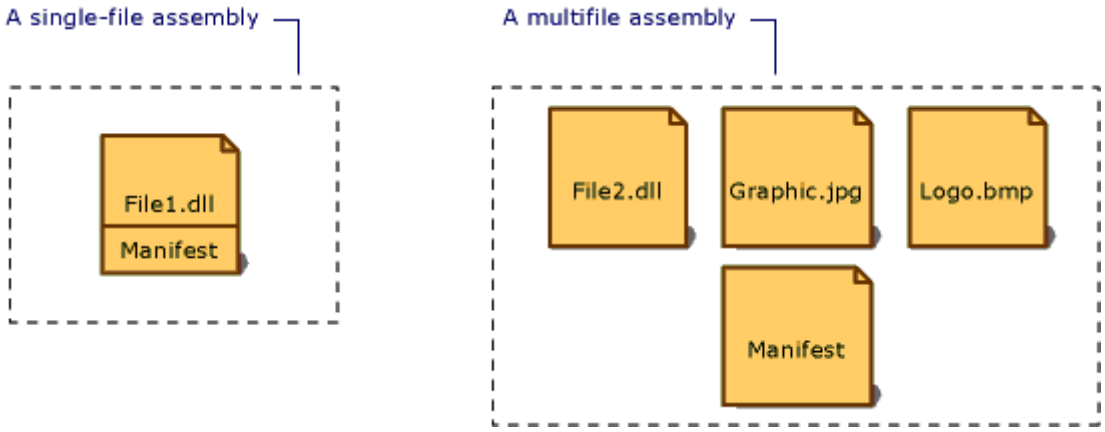
Provide product feedback

Assembly manifest

Article • 09/15/2021

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.



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.

Assembly manifest contents

The following table shows the information contained in the assembly manifest. The first four items: 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 Set assembly attributes.

See also

- Assembly contents
- Assembly versioning
- Create satellite assemblies
- Strong-named assemblies

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Assembly security considerations

Article • 03/30/2023

When you build an assembly, you can specify a set of permissions that the assembly requires to run. Whether certain permissions are granted or not granted to an assembly is based on evidence.

There are two distinct ways evidence is used:

- The input evidence is merged with the evidence gathered by the loader to create a final set of evidence used for policy resolution. The methods that use this semantic include **Assembly.Load**, **Assembly.LoadFrom**, and **Activator.CreateInstance**.
- The input evidence is used unaltered as the final set of evidence used for policy resolution. The methods that use this semantic include **Assembly.Load(byte[])** and **AppDomain.DefineDynamicAssembly()**.

  Optional permissions can be granted by the security policy set on the computer where the assembly will run. If you want your code to handle all potential security exceptions, you can do one of the following:

- Insert a permission request for all the permissions your code must have, and handle up front the load-time failure that occurs if the permissions are not granted.
- Do not use a permission request to obtain permissions your code might need, but be prepared to handle security exceptions if permissions are not granted.

> **Note**
>
> Security is a complex area, and you have many options to choose from. For more information, see **Key Security Concepts**.

At load time, the assembly's evidence is used as input to security policy. Security policy is established by the enterprise and the computer's administrator as well as by user policy settings, and determines the set of permissions that's granted to all managed code when executed. Security policy can be established for the publisher of the assembly (if it has a signing tool generated signature), for the Web site and zone (which was an Internet Explorer concept) that the assembly was downloaded from, or for the assembly's strong name. For example, a computer's administrator can establish security policy that allows all code downloaded from a Web site and signed by a given software

company to access a database on a computer, but does not grant access to write to the computer's disk.

Strong-named assemblies and signing tools

> ⚠ **Warning**
>
> Do not rely on strong names for security. They provide a unique identity only.

You can sign an assembly in two different but complementary ways: with a strong name or by using SignTool.exe (Sign Tool). Signing an assembly with a strong name adds public key encryption to the file containing the assembly manifest. Strong name signing helps to verify name uniqueness, prevent name spoofing, and provide callers with some identity when a reference is resolved.

No level of trust is associated with a strong name, which makes SignTool.exe (Sign Tool) important. The two signing tools require a publisher to prove its identity to a third-party authority and obtain a certificate. This certificate is then embedded in your file and can be used by an administrator to decide whether to trust the code's authenticity.

You can give both a strong name and a digital signature created using SignTool.exe (Sign Tool) to an assembly, or you can use either alone. The two signing tools can sign only one file at a time; for a multifile assembly, you sign the file that contains the assembly manifest. A strong name is stored in the file containing the assembly manifest, but a signature created using SignTool.exe (Sign Tool) is stored in a reserved slot in the portable executable (PE) file containing the assembly manifest. Signing of an assembly using SignTool.exe (Sign Tool) can be used (with or without a strong name) when you already have a trust hierarchy that relies on SignTool.exe (Sign Tool) generated signatures, or when your policy uses only the key portion and does not check a chain of trust.

> ⓘ **Note**
>
> When using both a strong name and a signing tool signature on an assembly, the strong name must be assigned first.

The common language runtime also performs a hash verification; the assembly manifest contains a list of all files that make up the assembly, including a hash of each file as it existed when the manifest was built. As each file is loaded, its contents are hashed and

compared with the hash value stored in the manifest. If the two hashes do not match, the assembly fails to load.

Strong naming and signing using SignTool.exe (Sign Tool) guarantee integrity through digital signatures and certificates. All the technologies mentioned, that is, hash verification, strong naming, and signing using SignTool.exe (Sign Tool), work together to ensure that the assembly has not been altered in any way.

See also

- Strong-named assemblies
- Assemblies in .NET
- SignTool.exe (Sign Tool)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Assembly versioning

Article • 10/01/2021

All versioning of assemblies that use the common language runtime is done at the assembly level. The specific version of an assembly and the versions of dependent assemblies are recorded in the assembly's manifest. The default version policy for the runtime is that applications run only with the versions they were built and tested with, unless overridden by explicit version policy in configuration files (the application configuration file, the publisher policy file, and the computer's administrator configuration file).

The runtime performs several steps to resolve an assembly binding request:

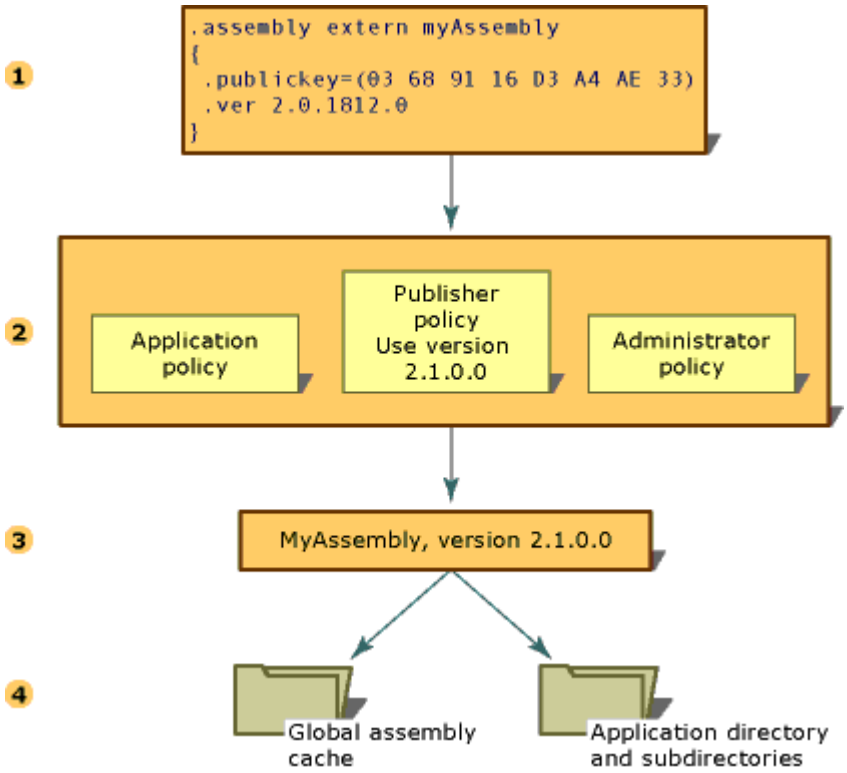
1. Checks the original assembly reference to determine the version of the assembly to be bound.

2. Checks for all applicable configuration files to apply version policy.

3. Determines the correct assembly from the original assembly reference and any redirection specified in the configuration files, and determines the version that should be bound to the calling assembly.

4. Checks the global assembly cache, codebases specified in configuration files, and then checks the application's directory and subdirectories using the probing rules explained in How the runtime locates assemblies.

The following illustration shows these steps:



For more information about configuring applications, see Configure apps. For more information about binding policy, see How the runtime locates assemblies.

Version information

Each assembly has two distinct ways of expressing version information:

- The assembly's version number, which, together with the assembly name and culture information, is part of the assembly's identity. This number is used by the runtime to enforce version policy and plays a key part in the type resolution process at run time.
- An informational version, which is a string that represents additional version information included for informational purposes only.

Assembly version number

Each assembly has a version number as part of its identity. As such, two assemblies that differ by version number are considered by the runtime to be completely different assemblies. This version number is physically represented as a four-part string with the following format:

<major version>.<minor version>.<build number>.<revision>

For example, version 1.5.1254.0 indicates 1 as the major version, 5 as the minor version, 1254 as the build number, and 0 as the revision number.

The version number is stored in the assembly manifest along with other identity information, including the assembly name and public key, as well as information on relationships and identities of other assemblies connected with the application.

When an assembly is built, the development tool records dependency information for each assembly that is referenced in the assembly manifest. The runtime uses these version numbers, in conjunction with configuration information set by an administrator, an application, or a publisher, to load the proper version of a referenced assembly.

The runtime distinguishes between regular and strong-named assemblies for the purposes of versioning. Version checking only occurs with strong-named assemblies.

For information about specifying version binding policies, see Configure apps. For information about how the runtime uses version information to find a particular assembly, see How the runtime locates assemblies.

Assembly informational version

The informational version is a string that attaches additional version information to an assembly for informational purposes only; this information is not used at run time. The text-based informational version corresponds to the product's marketing literature, packaging, or product name and is not used by the runtime. For example, an informational version could be "Common Language Runtime version 1.0" or "NET Control SP 2". On the Version tab of the file properties dialog in Microsoft Windows, this information appears in the item "Product Version".

> **Note**
>
> Although you can specify any text, a warning message appears on compilation if the string is not in the format used by the assembly version number, or if it is in that format but contains wildcards. This warning is harmless.

The informational version is represented using the custom attribute System.Reflection.AssemblyInformationalVersionAttribute. For more information about the informational version attribute, see Set assembly attributes.

See also

- How the runtime locates assemblies
- Configure apps
- Set assembly attributes

- Assemblies in .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

System.Version class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The Version class represents the version number of an assembly, operating system, or the common language runtime. Version numbers consist of two to four components: major, minor, build, and revision. The major and minor components are required; the build and revision components are optional, but the build component is required if the revision component is defined. All defined components must be integers greater than or equal to 0. The format of the version number is as follows (optional components are shown in square brackets):

major.minor[.*build*[.*revision*]]

The components are used by convention as follows:

- *Major*: Assemblies with the same name but different major versions are not interchangeable. A higher version number might indicate a major rewrite of a product where backward compatibility cannot be assumed.
- *Minor*: If the name and major version number on two assemblies are the same, but the minor version number is different, this indicates significant enhancement with the intention of backward compatibility. This higher minor version number might indicate a point release of a product or a fully backward-compatible new version of a product.
- *Build*: A difference in build number represents a recompilation of the same source. Different build numbers might be used when the processor, platform, or compiler changes.
- *Revision*: Assemblies with the same name, major, and minor version numbers but different revisions are intended to be fully interchangeable. A higher revision number might be used in a build that fixes a security hole in a previously released assembly.

Subsequent versions of an assembly that differ only by build or revision numbers are considered to be Hotfix updates of the prior version.

> **Important**

The value of **Version** properties that have not been explicitly assigned a value is undefined (-1).

The MajorRevision and MinorRevision properties enable you to identify a temporary version of your application that, for example, corrects a problem until you can release a permanent solution. Furthermore, the Windows NT operating system uses the MajorRevision property to encode the service pack number.

Assign version information to assemblies

Ordinarily, the Version class is not used to assign a version number to an assembly. Instead, the AssemblyVersionAttribute class is used to define an assembly's version, as illustrated by the example in this article.

Retrieve version information

Version objects are most frequently used to store version information about some system or application component (such as the operating system), the common language runtime, the current application's executable, or a particular assembly. The following examples illustrate some of the most common scenarios:

- Retrieving the operating system version. The following example uses the OperatingSystem.Version property to retrieve the version number of the operating system.

```
C#

// Get the operating system version.
OperatingSystem os = Environment.OSVersion;
Version ver = os.Version;
Console.WriteLine("Operating System: {0} ({1})", os.VersionString,
ver.ToString());
```

- Retrieving the version of the common language runtime. The following example uses the Environment.Version property to retrieve version information about the common language runtime.

```
C#

// Get the common language runtime version.
Version ver = Environment.Version;
Console.WriteLine("CLR Version {0}", ver.ToString());
```

- Retrieving the current application's assembly version. The following example uses the Assembly.GetEntryAssembly method to obtain a reference to an Assembly object that represents the application executable and then retrieves its assembly version number.

```
C#
```

```
using System;
using System.Reflection;

public class Example4
{
   public static void Main()
   {
      // Get the version of the executing assembly (that is, this
assembly).
      Assembly assem = Assembly.GetEntryAssembly();
      AssemblyName assemName = assem.GetName();
      Version ver = assemName.Version;
      Console.WriteLine("Application {0}, Version {1}", assemName.Name,
ver.ToString());
   }
}
```

- Retrieving the current assembly's assembly version. The following example uses the Type.Assembly property to obtain a reference to an Assembly object that represents the assembly that contains the application entry point, and then retrieves its version information.

```
C#
```

```
using System;
using System.Reflection;

public class Example3
{
   public static void Main()
   {
      // Get the version of the current assembly.
      Assembly assem = typeof(Example).Assembly;
      AssemblyName assemName = assem.GetName();
      Version ver = assemName.Version;
      Console.WriteLine("{0}, Version {1}", assemName.Name,
ver.ToString());
   }
}
```

- Retrieving the version of a specific assembly. The following example uses the Assembly.ReflectionOnlyLoadFrom method to obtain a reference to an Assembly

object that has a particular file name, and then retrieves its version information. Note that several other methods also exist to instantiate an Assembly object by file name or by strong name.

C#

```
using System;
using System.Reflection;

public class Example5
{
   public static void Main()
   {
      // Get the version of a specific assembly.
      string filename = @".\StringLibrary.dll";
      Assembly assem = Assembly.ReflectionOnlyLoadFrom(filename);
      AssemblyName assemName = assem.GetName();
      Version ver = assemName.Version;
      Console.WriteLine("{0}, Version {1}", assemName.Name,
ver.ToString());
   }
}
```

- Retrieving the Publish Version of a ClickOnce application. The following example uses the ApplicationDeployment.CurrentVersion property to display an application's Publish Version. Note that its successful execution requires the example's application identity to be set. This is handled automatically by the Visual Studio Publish Wizard.

C#

```
using System;
using System.Deployment.Application;

public class Example
{
   public static void Main()
   {
      Version ver =
ApplicationDeployment.CurrentDeployment.CurrentVersion;
      Console.WriteLine("ClickOnce Publish Version: {0}", ver);
   }
}
```

Important

> The Publish Version of an application for ClickOnce deployment is completely independent of its assembly version.

Compare version objects

You can use the CompareTo method to determine whether one Version object is earlier than, the same as, or later than a second Version object. The following example indicates that Version 2.1 is later than Version 2.0.

C#

```
Version v1 = new Version(2, 0);
Version v2 = new Version("2.1");
Console.Write("Version {0} is ", v1);
switch(v1.CompareTo(v2))
{
   case 0:
      Console.Write("the same as");
      break;
   case 1:
      Console.Write("later than");
      break;
   case -1:
      Console.Write("earlier than");
      break;
}
Console.WriteLine(" Version {0}.", v2);
// The example displays the following output:
//       Version 2.0 is earlier than Version 2.1.
```

For two versions to be equal, the major, minor, build, and revision numbers of the first Version object must be identical to those of the second Version object. If the build or revision number of a Version object is undefined, that Version object is considered to be earlier than a Version object whose build or revision number is equal to zero. The following example illustrates this by comparing three Version objects that have undefined version components.

C#

```
using System;

enum VersionTime {Earlier = -1, Same = 0, Later = 1 };

public class Example2
{
   public static void Main()
   {
```

```
      Version v1 = new Version(1, 1);
      Version v1a = new Version("1.1.0");
      ShowRelationship(v1, v1a);

      Version v1b = new Version(1, 1, 0, 0);
      ShowRelationship(v1b, v1a);
   }

   private static void ShowRelationship(Version v1, Version v2)
   {
      Console.WriteLine("Relationship of {0} to {1}: {2}",
                        v1, v2, (VersionTime) v1.CompareTo(v2));
   }
}
// The example displays the following output:
//       Relationship of 1.1 to 1.1.0: Earlier
//       Relationship of 1.1.0.0 to 1.1.0: Later
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Assemblies and side-by-side execution

Article • 09/15/2021

Side-by-side execution is the ability to store and execute multiple versions of an application or component on the same computer. This means that you can have multiple versions of the runtime, and multiple versions of applications and components that use a version of the runtime, on the same computer at the same time. Side-by-side execution gives you more control over what versions of a component an application binds to, and more control over what version of the runtime an application uses.

Support for side-by-side storage and execution of different versions of the same assembly is an integral part of strong naming and is built into the infrastructure of the runtime. Because the strong-named assembly's version number is part of its identity, the runtime can store multiple versions of the same assembly in the global assembly cache and load those assemblies at run time.

Although the runtime provides you with the ability to create side-by-side applications, side-by-side execution is not automatic. For more information on creating applications for side-by-side execution, see Guidelines for creating components for side-by-side execution.

See also

- How the runtime locates assemblies
- Assemblies in .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

.NET assembly file format

Article • 09/15/2021

.NET defines a binary file format, *assembly*, that is used to fully describe and contain .NET programs. Assemblies are used for the programs themselves as well as any dependent libraries. A .NET program can be executed as one or more assemblies, with no other required artifacts, beyond the appropriate .NET implementation. Native dependencies, including operating system APIs, are a separate concern and are not contained within the .NET assembly format, although they are sometimes described with this format (for example, WinRT).

> Each CLI component carries the metadata for declarations, implementations, and references specific to that component. Therefore, the component-specific metadata is referred to as component metadata, and the resulting component is said to be self-describing – from ECMA 335 I.9.1, Components and assemblies.

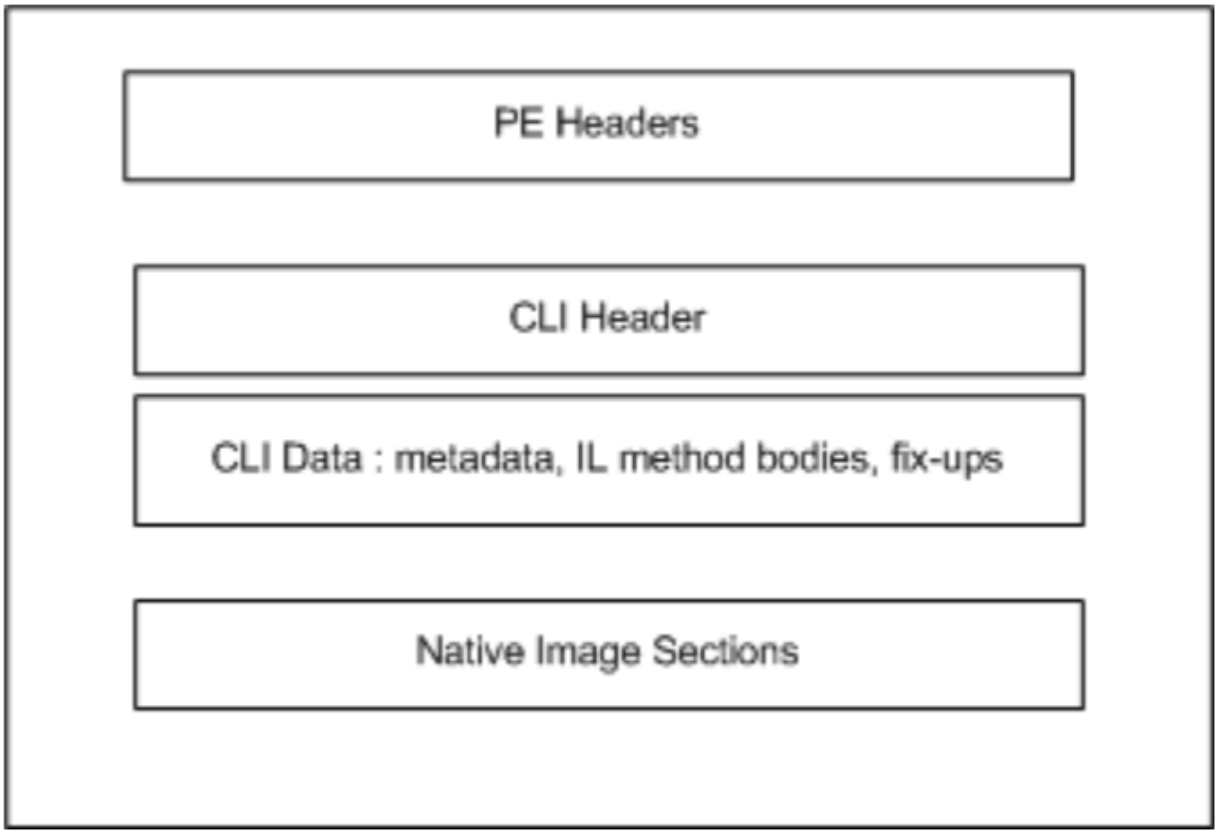
The format is fully specified and standardized as ECMA 335. All .NET compilers and runtimes use this format. The presence of a documented and infrequently updated binary format has been a major benefit (arguably a requirement) for interoperability. The format was last updated in a substantive way in 2005 (.NET Framework 2.0) to accommodate generics and processor architecture.

The format is CPU- and OS-agnostic. It has been used as part of .NET implementations that target many chips and CPUs. While the format itself has Windows heritage, it is implementable on any operating system. Its arguably most significant choice for OS interoperability is that most values are stored in little-endian format. It doesn't have a specific affinity to machine pointer size (for example, 32-bit, 64-bit).

The .NET assembly format is also very descriptive about the structure of a given program or library. It describes the internal components of an assembly, specifically assembly references and types defined and their internal structure. Tools or APIs can read and process this information for display or to make programmatic decisions.

Format

The .NET binary format is based on the Windows PE file format. In fact, .NET class libraries are conformant Windows PEs, and appear on first glance to be Windows dynamic link libraries (DLLs) or application executables (EXEs). This is a very useful characteristic on Windows, where they can masquerade as native executable binaries and get some of the same treatment (for example, OS load, PE tools).



Assembly Headers from ECMA 335 II.25.1, Structure of the runtime file format.

Process the assemblies

It is possible to write tools or APIs to process assemblies. Assembly information enables making programmatic decisions at run time, re-writing assemblies, providing API IntelliSense in an editor and generating documentation. System.Reflection, System.Reflection.MetadataLoadContext, and Mono.Cecil are good examples of tools that are frequently used for this purpose.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to use and debug assembly unloadability in .NET

Article • 11/13/2023

.NET (Core) introduced the ability to load and later unload a set of assemblies. In .NET Framework, custom app domains were used for this purpose, but .NET (Core) only supports a single default app domain.

Unloadability is supported through AssemblyLoadContext. You can load a set of assemblies into a collectible `AssemblyLoadContext`, execute methods in them or just inspect them using reflection, and finally unload the `AssemblyLoadContext`. That unloads the assemblies loaded into the `AssemblyLoadContext`.

There's one noteworthy difference between the unloading using `AssemblyLoadContext` and using AppDomains. With AppDomains, the unloading is forced. At unload time, all threads running in the target AppDomain are aborted, managed COM objects created in the target AppDomain are destroyed, and so on. With `AssemblyLoadContext`, the unload is "cooperative". Calling the AssemblyLoadContext.Unload method just initiates the unloading. The unloading finishes after:

- No threads have methods from the assemblies loaded into the `AssemblyLoadContext` on their call stacks.
- None of the types from the assemblies loaded into the `AssemblyLoadContext`, instances of those types, and the assemblies themselves are referenced by:
  - References outside of the `AssemblyLoadContext`, except for weak references (WeakReference or WeakReference<T>).
  - Strong garbage collector (GC) handles (GCHandleType.Normal or GCHandleType.Pinned) from both inside and outside of the `AssemblyLoadContext`.

Use collectible AssemblyLoadContext

This section contains a detailed step-by-step tutorial that shows a simple way to load a .NET (Core) application into a collectible `AssemblyLoadContext`, execute its entry point, and then unload it. You can find a complete sample at https://github.com/dotnet/samples/tree/main/core/tutorials/Unloading.

Create a collectible AssemblyLoadContext

Derive your class from the AssemblyLoadContext and override its AssemblyLoadContext.Load method. That method resolves references to all assemblies that are dependencies of assemblies loaded into that `AssemblyLoadContext`.

The following code is an example of the simplest custom `AssemblyLoadContext`:

C#

```
class TestAssemblyLoadContext : AssemblyLoadContext
{
    public TestAssemblyLoadContext() : base(isCollectible: true)
    {
    }

    protected override Assembly? Load(AssemblyName name)
    {
        return null;
    }
}
```

As you can see, the `Load` method returns `null`. That means that all the dependency assemblies are loaded into the default context, and the new context contains only the assemblies explicitly loaded into it.

If you want to load some or all of the dependencies into the `AssemblyLoadContext` too, you can use the `AssemblyDependencyResolver` in the `Load` method. The `AssemblyDependencyResolver` resolves the assembly names to absolute assembly file paths. The resolver uses the *.deps.json* file and assembly files in the directory of the main assembly loaded into the context.

C#

```
using System.Reflection;
using System.Runtime.Loader;

namespace complex
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public TestAssemblyLoadContext(string mainAssemblyToLoadPath) :
base(isCollectible: true)
        {
            _resolver = new
AssemblyDependencyResolver(mainAssemblyToLoadPath);
        }

        protected override Assembly? Load(AssemblyName name)
```

```
        {
            string? assemblyPath = _resolver.ResolveAssemblyToPath(name);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }
    }
}
```

Use a custom collectible AssemblyLoadContext

This section assumes the simpler version of the `TestAssemblyLoadContext` is being used.

You can create an instance of the custom `AssemblyLoadContext` and load an assembly into it as follows:

C#

```
var alc = new TestAssemblyLoadContext();
Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
```

For each of the assemblies referenced by the loaded assembly, the `TestAssemblyLoadContext.Load` method is called so that the `TestAssemblyLoadContext` can decide where to get the assembly from. In this case, it returns `null` to indicate that it should be loaded into the default context from locations that the runtime uses to load assemblies by default.

Now that an assembly was loaded, you can execute a method from it. Run the `Main` method:

C#

```
var args = new object[1] {new string[] {"Hello"}};
_ = a.EntryPoint?.Invoke(null, args);
```

After the `Main` method returns, you can initiate unloading by either calling the `Unload` method on the custom `AssemblyLoadContext` or removing the reference you have to the `AssemblyLoadContext`:

C#

```
alc.Unload();
```

This is sufficient to unload the test assembly. Next, you'll put all of this into a separate noninlineable method to ensure that the `TestAssemblyLoadContext`, `Assembly`, and `MethodInfo` (the `Assembly.EntryPoint`) can't be kept alive by stack slot references (real- or JIT-introduced locals). That could keep the `TestAssemblyLoadContext` alive and prevent the unload.

Also, return a weak reference to the `AssemblyLoadContext` so that you can use it later to detect unload completion.

C#

```
[MethodImpl(MethodImplOptions.NoInlining)]
static void ExecuteAndUnload(string assemblyPath, out WeakReference
alcWeakRef)
{
    var alc = new TestAssemblyLoadContext();
    Assembly a = alc.LoadFromAssemblyPath(assemblyPath);

    alcWeakRef = new WeakReference(alc, trackResurrection: true);

    var args = new object[1] {new string[] {"Hello"}};
    _ = a.EntryPoint?.Invoke(null, args);

    alc.Unload();
}
```

Now you can run this function to load, execute, and unload the assembly.

C#

```
WeakReference testAlcWeakRef;
ExecuteAndUnload("absolute/path/to/your/assembly", out testAlcWeakRef);
```

However, the unload doesn't complete immediately. As previously mentioned, it relies on the garbage collector to collect all the objects from the test assembly. In many cases, it isn't necessary to wait for the unload completion. However, there are cases where it's useful to know that the unload has finished. For example, you might want to delete the assembly file that was loaded into the custom `AssemblyLoadContext` from disk. In such a case, the following code snippet can be used. It triggers garbage collection and waits for pending finalizers in a loop until the weak reference to the custom `AssemblyLoadContext` is set to `null`, indicating the target object was collected. In most cases, just one pass through the loop is required. However, for more complex cases where objects created

by the code running in the `AssemblyLoadContext` have finalizers, more passes might be needed.

```
C#

for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
{
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

The Unloading event

In some cases, it might be necessary for the code loaded into a custom `AssemblyLoadContext` to perform some cleanup when the unloading is initiated. For example, it might need to stop threads or clean up strong GC handles. The `Unloading` event can be used in such cases. You can hook a handler that performs the necessary cleanup to this event.

Troubleshoot unloadability issues

Due to the cooperative nature of the unloading, it's easy to forget about references that might be keeping the stuff in a collectible `AssemblyLoadContext` alive and preventing unload. Here's a summary of entities (some of them nonobvious) that can hold the references:

- Regular references held from outside of the collectible `AssemblyLoadContext` that are stored in a stack slot or a processor register (method locals, either explicitly created by the user code or implicitly by the just-in-time (JIT) compiler), a static variable, or a strong (pinning) GC handle, and transitively pointing to:
  - An assembly loaded into the collectible `AssemblyLoadContext`.
  - A type from such an assembly.
  - An instance of a type from such an assembly.
- Threads running code from an assembly loaded into the collectible `AssemblyLoadContext`.
- Instances of custom, noncollectible `AssemblyLoadContext` types created inside of the collectible `AssemblyLoadContext`.
- Pending RegisteredWaitHandle instances with callbacks set to methods in the custom `AssemblyLoadContext`.

Tip

Object references that are stored in stack slots or processor registers and that could prevent unloading of an `AssemblyLoadContext` can occur in the following situations:

- When function call results are passed directly to another function, even though there is no user-created local variable.
- When the JIT compiler keeps a reference to an object that was available at some point in a method.

Debug unloading issues

Debugging issues with unloading can be tedious. You can get into situations where you don't know what can be holding an `AssemblyLoadContext` alive, but the unload fails. The best tool to help with that is WinDbg (or LLDB on Unix) with the SOS plugin. You need to find what's keeping a `LoaderAllocator` that belongs to the specific `AssemblyLoadContext` alive. The SOS plugin lets you look at GC heap objects, their hierarchies, and roots.

To load the SOS plugin into the debugger, enter one of the following commands in the debugger command line.

In WinDbg (if it's not already loaded):

Console

```
.loadby sos coreclr
```

In LLDB:

Console

```
plugin load /path/to/libsosplugin.so
```

Now you'll debug an example program that has problems with unloading. The source code is available in the Example source code section. When you run it under WinDbg, the program breaks into the debugger right after attempting to check for the unload success. You can then start looking for the culprits.

Tip

If you debug using LLDB on Unix, the SOS commands in the following examples don't have the `!` in front of them.

Console

```
!dumpheap -type LoaderAllocator
```

This command dumps all objects with a type name containing `LoaderAllocator` that are in the GC heap. Here's an example:

Console

```
         Address               MT     Size
000002b78000ce40 00007ffadc93a288       48
000002b78000ceb0 00007ffadc93a218       24

Statistics:
              MT    Count    TotalSize Class Name
00007ffadc93a218        1           24
System.Reflection.LoaderAllocatorScout
00007ffadc93a288        1           48 System.Reflection.LoaderAllocator
Total 2 objects
```

In the "Statistics:" part, check the `MT` (`MethodTable`) that belongs to the `System.Reflection.LoaderAllocator`, which is the object you care about. Then, in the list at the beginning, find the entry with `MT` that matches that one, and get the address of the object itself. In this case, it's "000002b78000ce40".

Now that you know the address of the `LoaderAllocator` object, you can use another command to find its GC roots:

Console

```
!gcroot 0x000002b78000ce40
```

This command dumps the chain of object references that lead to the `LoaderAllocator` instance. The list starts with the root, which is the entity that keeps the `LoaderAllocator` alive and thus is the core of the problem. The root can be a stack slot, a processor register, a GC handle, or a static variable.

Here's an example of the output of the `gcroot` command:

Console

```
Thread 4ac:
    000000cf9499dd20 00007ffa7d0236bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
        rbp-20: 000000cf9499dd90
            ->  000002b78000d328 System.Reflection.RuntimeMethodInfo
            ->  000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
            ->  000002b78000d1d0 System.RuntimeType
            ->  000002b78000ce40 System.Reflection.LoaderAllocator

HandleTable:
    000002b7f8a81198 (strong handle)
    -> 000002b78000d948 test.Test
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

    000002b7f8a815f8 (pinned handle)
    -> 000002b790001038 System.Object[]
    -> 000002b78000d390 example.TestInfo
    -> 000002b78000d328 System.Reflection.RuntimeMethodInfo
    -> 000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
    -> 000002b78000d1d0 System.RuntimeType
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

Found 3 roots.
```

The next step is to figure out where the root is located so you can fix it. The easiest case is when the root is a stack slot or a processor register. In that case, the `gcroot` shows the name of the function whose frame contains the root and the thread executing that function. The difficult case is when the root is a static variable or a GC handle.

In the previous example, the first root is a local of type `System.Reflection.RuntimeMethodInfo` stored in the frame of the function `example.Program.Main(System.String[])` at address `rbp-20` (`rbp` is the processor register `rbp` and -20 is a hexadecimal offset from that register).

The second root is a normal (strong) `GCHandle` that holds a reference to an instance of the `test.Test` class.

The third root is a pinned `GCHandle`. This one is actually a static variable, but unfortunately, there's no way to tell. Statics for reference types are stored in a managed object array in internal runtime structures.

Another case that can prevent unloading of an `AssemblyLoadContext` is when a thread has a frame of a method from an assembly loaded into the `AssemblyLoadContext` on its stack. You can check that by dumping managed call stacks of all threads:

Console

```
~*e !clrstack
```

The command means "apply to all threads the `!clrstack` command". The following is the output of that command for the example. Unfortunately, LLDB on Unix doesn't have any way to apply a command to all threads, so you must manually switch threads and repeat the `clrstack` command. Ignore all threads where the debugger says "Unable to walk the managed stack".

Console

```
OS Thread Id: 0x6ba8 (0)
        Child SP               IP Call Site
0000001fc697d5c8 00007ffb50d9de12 [HelperMethodFrame: 0000001fc697d5c8]
System.Diagnostics.Debugger.BreakInternal()
0000001fc697d6d0 00007ffa864765fa System.Diagnostics.Debugger.Break()
0000001fc697d700 00007ffa864736bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
0000001fc697d998 00007ffae5fdc1e3 [GCFrame: 0000001fc697d998]
0000001fc697df28 00007ffae5fdc1e3 [GCFrame: 0000001fc697df28]
OS Thread Id: 0x2ae4 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x61a4 (2)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x7fdc (3)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5390 (4)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5ec8 (5)
        Child SP               IP Call Site
0000001fc70ff6e0 00007ffb5437f6e4 [DebuggerU2MCatchHandlerFrame:
0000001fc70ff6e0]
OS Thread Id: 0x4624 (6)
        Child SP               IP Call Site
GetFrameContext failed: 1
0000000000000000 0000000000000000
OS Thread Id: 0x60bc (7)
        Child SP               IP Call Site
0000001fc727f158 00007ffb5437fce4 [HelperMethodFrame: 0000001fc727f158]
System.Threading.Thread.SleepInternal(Int32)
```

```
0000001fc727f260 00007ffb37ea7c2b System.Threading.Thread.Sleep(Int32)
0000001fc727f290 00007ffa865005b3 test.Program.ThreadProc()
[E:\unloadability\test\Program.cs @ 17]
0000001fc727f2c0 00007ffb37ea6a5b
System.Threading.Thread.ThreadMain_ThreadStart()
0000001fc727f2f0 00007ffadbc4cbe3
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionCont
ext, System.Threading.ContextCallback, System.Object)
0000001fc727f568 00007ffae5fdc1e3 [GCFrame: 0000001fc727f568]
0000001fc727f7f0 00007ffae5fdc1e3 [DebuggerU2MCatchHandlerFrame:
0000001fc727f7f0]
```

As you can see, the last thread has `test.Program.ThreadProc()`. This is a function from the assembly loaded into the `AssemblyLoadContext`, and so it keeps the `AssemblyLoadContext` alive.

Example source code

The following code that contains unloadability issues is used in the previous debugging example.

Main testing program

C#

```
using System;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.Loader;

namespace example
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        public TestAssemblyLoadContext() : base(true)
        {
        }
        protected override Assembly? Load(AssemblyName name)
        {
            return null;
        }
    }

    class TestInfo
    {
        public TestInfo(MethodInfo? mi)
        {
            _entryPoint = mi;
        }
```

```
        MethodInfo? _entryPoint;
    }

    class Program
    {
        static TestInfo? entryPoint;

        [MethodImpl(MethodImplOptions.NoInlining)]
        static int ExecuteAndUnload(string assemblyPath, out WeakReference
testAlcWeakRef, out MethodInfo? testEntryPoint)
        {
            var alc = new TestAssemblyLoadContext();
            testAlcWeakRef = new WeakReference(alc);

            Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
            if (a == null)
            {
                testEntryPoint = null;
                Console.WriteLine("Loading the test assembly failed");
                return -1;
            }

            var args = new object[1] {new string[] {"Hello"}};

            // Issue preventing unloading #1 - we keep MethodInfo of a
method
            // for an assembly loaded into the TestAssemblyLoadContext in a
static variable.
            entryPoint = new TestInfo(a.EntryPoint);
            testEntryPoint = a.EntryPoint;

            var oResult = a.EntryPoint?.Invoke(null, args);
            alc.Unload();
            return (oResult is int result) ? result : -1;
        }

        static void Main(string[] args)
        {
            WeakReference testAlcWeakRef;
            // Issue preventing unloading #2 - we keep MethodInfo of a
method for an assembly loaded into the TestAssemblyLoadContext in a local
variable
            MethodInfo? testEntryPoint;
            int result = ExecuteAndUnload(@"absolute/path/to/test.dll", out
testAlcWeakRef, out testEntryPoint);

            for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
            {
                GC.Collect();
                GC.WaitForPendingFinalizers();
            }

            System.Diagnostics.Debugger.Break();
```

```
            Console.WriteLine($"Test completed, result={result}, entryPoint:
{testEntryPoint} unload success: {!testAlcWeakRef.IsAlive}");
        }
    }
}
```

Program loaded into the TestAssemblyLoadContext

The following code represents the *test.dll* passed to the `ExecuteAndUnload` method in the main testing program.

C#

```
using System;
using System.Runtime.InteropServices;
using System.Threading;

namespace test
{
    class Test
    {
    }

    class Program
    {
        public static void ThreadProc()
        {
            // Issue preventing unloading #4 - a thread running method
inside of the TestAssemblyLoadContext at the unload time
            Thread.Sleep(Timeout.Infinite);
        }

        static GCHandle handle;
        static int Main(string[] args)
        {
            // Issue preventing unloading #3 - normal GC handle
            handle = GCHandle.Alloc(new Test());
            Thread t = new Thread(new ThreadStart(ThreadProc));
            t.IsBackground = true;
            t.Start();
            Console.WriteLine($"Hello from the test: args[0] = {args[0]}");

            return 1;
        }
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Reference assemblies

Article • 09/15/2021

Reference assemblies are a special type of assembly that contain only the minimum amount of metadata required to represent the library's public API surface. They include declarations for all members that are significant when referencing an assembly in build tools, but exclude all member implementations and declarations of private members that have no observable impact on their API contract. In contrast, regular assemblies are called *implementation assemblies*.

Reference assemblies can't be loaded for execution, but they can be passed as compiler input in the same way as implementation assemblies. Reference assemblies are usually distributed with the Software Development Kit (SDK) of a particular platform or library.

Using a reference assembly enables developers to build programs that target a specific library version without having the full implementation assembly for that version. Suppose, you have only the latest version of some library on your machine, but you want to build a program that targets an earlier version of that library. If you compile directly against the implementation assembly, you might inadvertently use API members that aren't available in the earlier version. You'll only find this mistake when testing the program on the target machine. If you compile against the reference assembly for the earlier version, you'll immediately get a compile-time error.

A reference assembly can also represent a contract, that is, a set of APIs that don't correspond to the concrete implementation assembly. Such reference assemblies, called the *contract assembly*, can be used to target multiple platforms that support the same set of APIs. For example, .NET Standard provides the contract assembly, *netstandard.dll*, that represents the set of common APIs shared between different .NET platforms. The implementations of these APIs are contained in different assemblies on different platforms, such as *mscorlib.dll* on .NET Framework or *System.Private.CoreLib.dll* on .NET Core. A library that targets .NET Standard can run on all platforms that support .NET Standard.

Using reference assemblies

To use certain APIs from your project, you must add references to their assemblies. You can add references to either implementation assemblies or to reference assemblies. It's recommended you use reference assemblies whenever they're available. Doing so ensures that you're using only the supported API members in the target version, meant

to be used by API designers. Using the reference assembly ensures you're not taking a dependency on implementation details.

Reference assemblies for the .NET Framework libraries are distributed with targeting packs. You can obtain them by downloading a standalone installer or by selecting a component in Visual Studio installer. For more information, see Install the .NET Framework for developers. For .NET Core and .NET Standard, reference assemblies are automatically downloaded as necessary (via NuGet) and referenced. For .NET Core 3.0 and higher, the reference assemblies for the core framework are in the Microsoft.NETCore.App.Ref package (the Microsoft.NETCore.App package is used instead for versions before 3.0).

When you add references to .NET Framework assemblies in Visual Studio using the **Add reference** dialog, you select an assembly from the list, and Visual Studio automatically finds reference assemblies that correspond to the target framework version selected in your project. The same applies to adding references directly into MSBuild project using the Reference project item: you only need to specify the assembly name, not the full file path. When you add references to these assemblies in the command line by using the `-reference` compiler option (in C# and in Visual Basic) or by using the Compilation.AddReferences method in the Roslyn API, you must manually specify reference assembly files for the correct target platform version. .NET Framework reference assembly files are located in the *%ProgramFiles(x86)%\Reference Assemblies\Microsoft\Framework\.NETFramework* directory. For .NET Core, you can force publish operation to copy reference assemblies for your target platform into the *publish/refs* subdirectory of your output directory by setting the `PreserveCompilationContext` project property to `true`. Then you can pass these reference assembly files to the compiler. Using `DependencyContext` from Microsoft.Extensions.DependencyModel package can help locate their paths.

Because they contain no implementation, reference assemblies can't be loaded for execution. Trying to do so results in a System.BadImageFormatException. If you want to examine the contents of a reference assembly, you can load it into the reflection-only context in .NET Framework (using the Assembly.ReflectionOnlyLoad method) or into the MetadataLoadContext in .NET Core.

Generating reference assemblies

Generating reference assemblies for your libraries can be useful when your library consumers need to build their programs against many different versions of the library. Distributing implementation assemblies for all these versions might be impractical

because of their large size. Reference assemblies are smaller in size, and distributing them as a part of your library's SDK reduces download size and saves disk space.

IDEs and build tools also can take advantage of reference assemblies to reduce build times in case of large solutions consisting of multiple class libraries. Usually, in incremental build scenarios a project is rebuilt when any of its input files are changed, including the assemblies it depends on. The implementation assembly changes whenever the programmer changes the implementation of any member. The reference assembly only changes when its public API is affected. So, using the reference assembly as an input file instead of the implementation assembly allows skipping the build of the dependent project in some cases.

You can generate reference assemblies:

- In an MSBuild project, by using the ProduceReferenceAssembly project property.
- When compiling program from command line, by specifying `-refonly` (C# / Visual Basic) or `-refout` (C# / Visual Basic) compiler options.
- When using the Roslyn API, by setting EmitOptions.EmitMetadataOnly to `true` and EmitOptions.IncludePrivateMembers to `false` in an object passed to the Compilation.Emit method.

If you want to distribute reference assemblies with NuGet packages, you must include them in the *ref* subdirectory under the package directory instead of in the *lib* subdirectory used for implementation assemblies.

Reference assemblies structure

Reference assemblies are an expansion of the related concept, *metadata-only assemblies*. Metadata-only assemblies have their method bodies replaced with a single `throw null` body, but include all members except anonymous types. The reason for using `throw null` bodies (as opposed to no bodies) is so that **PEVerify** can run and pass (thus validating the completeness of the metadata).

Reference assemblies further remove metadata (private members) from metadata-only assemblies:

- A reference assembly only has references for what it needs in the API surface. The real assembly may have additional references related to specific implementations. For instance, the reference assembly for `class C { private void M() { dynamic d = 1; ... } }` doesn't reference any types required for `dynamic`.
- Private function-members (methods, properties, and events) are removed in cases where their removal doesn't observably impact compilation. If there are no

InternalsVisibleTo attributes, internal function members are also removed.

The metadata in reference assemblies continues to keep the following information:

- All types, including private and nested types.
- All attributes, even internal ones.
- All virtual methods.
- Explicit interface implementations.
- Explicitly implemented properties and events, because their accessors are virtual.
- All fields of structures.

Reference assemblies include an assembly-level ReferenceAssembly attribute. This attribute may be specified in source; then the compiler won't need to synthesize it. Because of this attribute, runtimes will refuse to load reference assemblies for execution (but they can be loaded in reflection-only mode).

Exact reference assembly structure details depend on the compiler version. Newer versions may choose to exclude more metadata if it's determined as not affecting the public API surface.

> **Note**
>
> Information in this section is applicable only to reference assemblies generated by Roslyn compilers starting from C# version 7.1 or Visual Basic version 15.3. The structure of reference assemblies for .NET Framework and .NET Core libraries can differ in some details, because they use their own mechanism of generating reference assemblies. For example, they might have totally empty method bodies instead of the `throw null` body. But the general principle still applies: they don't have usable method implementations and contain metadata only for members that have an observable impact from a public API perspective.

See also

- Assemblies in .NET
- Framework targeting overview
- How to: Add or remove references by using the Reference Manager

Collaborate with us on GitHub

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Resolve assembly loads

Article • 09/15/2021

.NET provides the AppDomain.AssemblyResolve event for applications that require greater control over assembly loading. By handling this event, your application can load an assembly into the load context from outside the normal probing paths, select which of several assembly versions to load, emit a dynamic assembly and return it, and so on. This topic provides guidance for handling the AssemblyResolve event.

> **Note**
>
> For resolving assembly loads in the reflection-only context, use the **AppDomain.ReflectionOnlyAssemblyResolve** event instead.

How the AssemblyResolve event works

When you register a handler for the AssemblyResolve event, the handler is invoked whenever the runtime fails to bind to an assembly by name. For example, calling the following methods from user code can cause the AssemblyResolve event to be raised:

- An AppDomain.Load method overload or Assembly.Load method overload whose first argument is a string that represents the display name of the assembly to load (that is, the string returned by the Assembly.FullName property).
- An AppDomain.Load method overload or Assembly.Load method overload whose first argument is an AssemblyName object that identifies the assembly to load.
- An Assembly.LoadWithPartialName method overload.
- An AppDomain.CreateInstance or AppDomain.CreateInstanceAndUnwrap method overload that instantiates an object in another application domain.

What the event handler does

The handler for the AssemblyResolve event receives the display name of the assembly to be loaded, in the ResolveEventArgs.Name property. If the handler does not recognize the assembly name, it returns `null` (C#), `Nothing` (Visual Basic), or `nullptr` (Visual C++).

If the handler recognizes the assembly name, it can load and return an assembly that satisfies the request. The following list describes some sample scenarios.

- If the handler knows the location of a version of the assembly, it can load the assembly by using the Assembly.LoadFrom or Assembly.LoadFile method, and can return the loaded assembly if successful.
- If the handler has access to a database of assemblies stored as byte arrays, it can load a byte array by using one of the Assembly.Load method overloads that take a byte array.
- The handler can generate a dynamic assembly and return it.

> **Note**
>
> The handler must load the assembly into the load-from context, into the load context, or without context. If the handler loads the assembly into the reflection-only context by using the **Assembly.ReflectionOnlyLoad** or the **Assembly.ReflectionOnlyLoadFrom** method, the load attempt that raised the **AssemblyResolve** event fails.

It is the responsibility of the event handler to return a suitable assembly. The handler can parse the display name of the requested assembly by passing the ResolveEventArgs.Name property value to the AssemblyName(String) constructor. Beginning with the .NET Framework 4, the handler can use the ResolveEventArgs.RequestingAssembly property to determine whether the current request is a dependency of another assembly. This information can help identify an assembly that will satisfy the dependency.

The event handler can return a different version of the assembly than the version that was requested.

In most cases, the assembly that is returned by the handler appears in the load context, regardless of the context the handler loads it into. For example, if the handler uses the Assembly.LoadFrom method to load an assembly into the load-from context, the assembly appears in the load context when the handler returns it. However, in the following case the assembly appears without context when the handler returns it:

- The handler loads an assembly without context.
- The ResolveEventArgs.RequestingAssembly property is not null.
- The requesting assembly (that is, the assembly that is returned by the ResolveEventArgs.RequestingAssembly property) was loaded without context.

For information about contexts, see the Assembly.LoadFrom(String) method overload.

Multiple versions of the same assembly can be loaded into the same application domain. This practice is not recommended, because it can lead to type assignment problems. See Best practices for assembly loading.

What the event handler should not do

The primary rule for handling the AssemblyResolve event is that you should not try to return an assembly you do not recognize. When you write the handler, you should know which assemblies might cause the event to be raised. Your handler should return null for other assemblies.

> **Important**
>
> Beginning with the .NET Framework 4, the **AssemblyResolve** event is raised for satellite assemblies. This change affects an event handler that was written for an earlier version of the .NET Framework, if the handler tries to resolve all assembly load requests. Event handlers that ignore assemblies they do not recognize are not affected by this change: They return `null`, and normal fallback mechanisms are followed.

When loading an assembly, the event handler must not use any of the AppDomain.Load or Assembly.Load method overloads that can cause the AssemblyResolve event to be raised recursively, because this can lead to a stack overflow. (See the list provided earlier in this topic.) This happens even if you provide exception handling for the load request, because no exception is thrown until all event handlers have returned. Thus, the following code results in a stack overflow if `MyAssembly` is not found:

C#

```
using System;
using System.Reflection;

class BadExample
{
    static void Main()
    {
        AppDomain ad = AppDomain.CreateDomain("Test");
        ad.AssemblyResolve += MyHandler;

        try
        {
            object obj = ad.CreateInstanceAndUnwrap(
                "MyAssembly, version=1.2.3.4, culture=neutral,
publicKeyToken=null",
```

```
                "MyType");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }

    static Assembly MyHandler(object source, ResolveEventArgs e)
    {
        Console.WriteLine("Resolving {0}", e.Name);
        // DO NOT DO THIS: This causes a StackOverflowException
        return Assembly.Load(e.Name);
    }
}

/* This example produces output similar to the following:

Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
...
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null

Process is terminated due to StackOverflowException.
 */
```

The correct way to handle AssemblyResolve

When resolving assemblies from the AssemblyResolve event handler, a StackOverflowException will eventually be thrown if the handler uses the Assembly.Load or AppDomain.Load method calls. Instead, use LoadFile or LoadFrom methods, as they do not raise the `AssemblyResolve` event.

Imagine that `MyAssembly.dll` is located near the executing assembly, in a known location, it can be resolved using `Assembly.LoadFile` given the path to the assembly.

C#

```
using System;
using System.IO;
using System.Reflection;

class CorrectExample
{
    static void Main()
    {
        AppDomain ad = AppDomain.CreateDomain("Test");
        ad.AssemblyResolve += MyHandler;
```

```
        try
        {
            object obj = ad.CreateInstanceAndUnwrap(
                "MyAssembly, version=1.2.3.4, culture=neutral,
publicKeyToken=null",
                "MyType");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }

    static Assembly MyHandler(object source, ResolveEventArgs e)
    {
        Console.WriteLine("Resolving {0}", e.Name);

        var path = Path.GetFullPath("../../MyAssembly.dll");
        return Assembly.LoadFile(path);
     }
}
```

See also

- Best practices for assembly loading
- Use application domains

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Create assemblies

Article • 09/15/2021

You can create single-file or multifile assemblies using an IDE, such as Visual Studio, or the compilers and tools provided by the Windows SDK. The simplest assembly is a single file that has a simple name and is loaded into a single application domain. This assembly cannot be referenced by other assemblies outside the application directory and does not undergo version checking. To uninstall the application made up of the assembly, you simply delete the directory where it resides. For many developers, an assembly with these features is all that is needed to deploy an application.

You can create a multifile assembly from several code modules and resource files. You can also create an assembly that can be shared by multiple applications. A shared assembly must have a strong name and can be deployed in the global assembly cache.

You have several options when grouping code modules and resources into assemblies, depending on the following factors:

- Versioning

  Group modules that should have the same version information.

- Deployment

  Group code modules and resources that support your model of deployment.

- Reuse

  Group modules if they can be logically used together for some purpose. For example, an assembly consisting of types and classes used infrequently for program maintenance can be put in the same assembly. In addition, types that you intend to share with multiple applications should be grouped into an assembly and the assembly should be signed with a strong name.

- Security

  Group modules containing types that require the same security permissions.

- Scoping

  Group modules containing types whose visibility should be restricted to the same assembly.

There are special considerations when making common language runtime assemblies available to unmanaged COM applications. For more information about working with unmanaged code, see Expose .NET Framework components to COM.

See also

- Assembly versioning
- How to: Build a single-file assembly
- How to: Build a multifile assembly
- How the runtime locates assemblies
- Multifile assemblies

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Assembly names

Article • 09/15/2021

An assembly's name is stored in metadata and has a significant impact on the assembly's scope and use by an application. A strong-named assembly has a fully qualified name that includes the assembly's name, culture, public key, version number, and, optionally, processor architecture. Use the FullName property to obtain the fully qualified name, frequently referred to as the display name, for loaded assemblies.

The runtime uses the name information to locate the assembly and differentiate it from other assemblies with the same name. For example, a strong-named assembly called `myTypes` could have the following fully qualified name:

```
myTypes, Version=1.0.1234.0, Culture=en-US,
PublicKeyToken=b77a5c561934e089c, ProcessorArchitecture=msil
```

In this example, the fully qualified name indicates that the `myTypes` assembly has a strong name with a public key token, has the culture value for United States English, and has a version number of 1.0.1234.0. Its processor architecture is `msil`, which means that it will be just-in-time (JIT)-compiled to 32-bit code or 64-bit code depending on the operating system and processor.

> **Tip**
>
> The `ProcessorArchitecture` information allows processor-specific versions of assemblies. You can create versions of an assembly whose identity differs only by processor architecture, for example 32-bit and 64-bit processor-specific versions. Processor architecture is not required for strong names. For more information, see **AssemblyName.ProcessorArchitecture**.

Code that requests types in an assembly must use a fully qualified assembly name. This is called fully qualified binding. Partial binding, which specifies only an assembly name, is not permitted when referencing assemblies in .NET Framework.

All assembly references to assemblies that make up .NET Framework must also contain the fully qualified name of the assembly. For example, a reference to the System.Data .NET Framework assembly for version 1.0 would include:

```
System.data, version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
```

The version corresponds to the version number of all .NET Framework assemblies that shipped with .NET Framework version 1.0. For .NET Framework assemblies, the culture value is always neutral, and the public key is the same as shown in the above example.

For example, to add an assembly reference in a configuration file to set up a trace listener, you would include the fully qualified name of the system .NET Framework assembly:

XML

```xml
<add name="myListener" type="System.Diagnostics.TextWriterTraceListener,
System, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" initializeData="c:\myListener.log" />
```

> **Note**
>
> The runtime treats assembly names as case-insensitive when binding to an assembly, but preserves whatever case is used in an assembly name. Several tools in the Windows SDK handle assembly names as case-sensitive. For best results, manage assembly names as though they were case-sensitive.

Name application components

The runtime does not consider the file name when determining an assembly's identity. The assembly identity, which consists of the assembly name, version, culture, and strong name, must be clear to the runtime.

For example, if you have an assembly called *myAssembly.exe* that references an assembly called *myAssembly.dll*, binding occurs correctly if you execute *myAssembly.exe*. However, if another application executes *myAssembly.exe* using the method AppDomain.ExecuteAssembly, the runtime determines that `myAssembly` is already loaded when *myAssembly.exe* requests binding to `myAssembly`. In this case, *myAssembly.dll* is never loaded. Because *myAssembly.exe* does not contain the requested type, a TypeLoadException occurs.

To avoid this problem, make sure the assemblies that make up your application do not have the same assembly name or place assemblies with the same name in different directories.

> **Note**
>
> In .NET Framework, if you put a strong-named assembly in the global assembly cache, the assembly's file name must match the assembly name, not including the file name extension, such as *.exe* or *.dll*. For example, if the file name of an assembly is *myAssembly.dll*, the assembly name must be `myAssembly`. Private assemblies deployed only in the root application directory can have an assembly name that is different from the file name.

See also

- How to: Determine an assembly's fully qualified name
- Create assemblies
- Strong-named assemblies
- Global assembly cache
- How the runtime locates assemblies

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to: Find an assembly's fully qualified name

Article • 09/15/2021

To discover the fully qualified name of a .NET Framework assembly in the global assembly cache, use the Global Assembly Cache tool (Gacutil.exe). See How to: View the contents of the global assembly cache.

For .NET Core assemblies, and for .NET Framework assemblies that aren't in the global assembly cache, you can get the fully qualified assembly name in a number of ways:

- You can use code to output the information to the console or to a variable, or you can use the Ildasm.exe (IL Disassembler) to examine the assembly's metadata, which contains the fully qualified name.
- If the assembly is already loaded by the application, you can retrieve the value of the Assembly.FullName property to get the fully qualified name. You can use the Assembly property of a Type defined in that assembly to retrieve a reference to the Assembly object. The example provides an illustration.
- If you know the assembly's file system path, you can call the `static` (C#) or `Shared` (Visual Basic) AssemblyName.GetAssemblyName method to get the fully qualified assembly name. The following is a simple example.

  C#

  ```
  using System;
  using System.Reflection;

  public class Example
  {
     public static void Main()
     {

  Console.WriteLine(AssemblyName.GetAssemblyName(@".\UtilityLibrary.dll")
  );
     }
  }
  // The example displays output like the following:
  //    UtilityLibrary, Version=1.1.0.0, Culture=neutral,
  PublicKeyToken=null
  ```

- You can use the Ildasm.exe (IL Disassembler) to examine the assembly's metadata, which contains the fully qualified name.

For more information about setting assembly attributes such as version, culture, and assembly name, see Set assembly attributes. For more information about giving an assembly a strong name, see Create and use strong-named assemblies.

Example

The following example shows how to display the fully qualified name of an assembly containing a specified class to the console. It uses the Type.Assembly property to retrieve a reference to an assembly from a type that's defined in that assembly.

C#

```
using System;
using System.Reflection;

class asmname
{
    public static void Main()
    {
        Type t = typeof(System.Data.DataSet);
        string s = t.Assembly.FullName.ToString();
        Console.WriteLine("The fully qualified assembly name " +
            "containing the specified class is {0}.", s);
    }
}
```

See also

- Assembly names
- Create assemblies
- Create and use strong-named assemblies
- Global assembly cache
- How the runtime locates assemblies

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

Assembly location

Article • 09/15/2021

An assembly's location determines whether the common language runtime can locate it when referenced, and can also determine whether the assembly can be shared with other assemblies. You can deploy an assembly in the following locations:

- The application's directory or subdirectories.

  This is the most common location for deploying an assembly. The subdirectories of an application's root directory can be based on language or culture. If an assembly has information in the culture attribute, it must be in a subdirectory under the application directory with that culture's name.

- The global assembly cache.

  This is a machine-wide code cache that is installed wherever the common language runtime is installed. In most cases, if you intend to share an assembly with multiple applications, you should deploy it into the global assembly cache.

- On an HTTP server.

  An assembly deployed on an HTTP server must have a strong name; you point to the assembly in the codebase section of the application's configuration file.

See also

- Create assemblies
- Global assembly cache
- How the runtime locates assemblies

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Set assembly attributes in code

Article • 01/18/2023

Assembly attributes are values that provide information about an assembly. They're usually set in an *AssemblyInfo.cs* file. The attributes are divided into the following sets of information:

- Assembly identity attributes
- Informational attributes
- Assembly manifest attributes
- Strong name attributes

This article is scoped to adding assembly attributes from code. For information on adding assembly attributes to projects (not in code), see Set assembly attributes in a project file.

Assembly identity attributes

Three attributes, together with a strong name (if applicable), determine the identity of an assembly: name, version, and culture. These attributes form the full name of the assembly and are required when referencing the assembly in code. You can use attributes to set an assembly's version and culture. The compiler or the Assembly Linker (Al.exe) sets the name value when the assembly is created, based on the file containing the assembly manifest.

The following table describes the version and culture attributes.

Expand table

Assembly identity attribute	Description
AssemblyCultureAttribute	Enumerated field indicating the culture that the assembly supports. An assembly can also specify culture independence, indicating that it contains the resources for the default culture. **Note:** The runtime treats any assembly that does not have the culture attribute set to null as a satellite assembly. Such assemblies are subject to satellite assembly binding rules. For more information, see How the runtime locates assemblies.
AssemblyFlagsAttribute	Value that sets assembly attributes, such as whether the assembly can be run side by side.

Assembly identity attribute	Description
AssemblyVersionAttribute	Numeric value in the format *major.minor.build.revision* (for example, 2.4.0.0). The common language runtime uses this value to perform binding operations in strong-named assemblies. **Note:** If the AssemblyInformationalVersionAttribute attribute is not applied to an assembly, the version number specified by the AssemblyVersionAttribute attribute is used by the Application.ProductVersion, Application.UserAppDataPath, and Application.UserAppDataRegistry properties.

The following code example shows how to apply the version and culture attributes to an assembly.

C#

```
// Set version number for the assembly.
[assembly:AssemblyVersionAttribute("4.3.2.1")]
// Set culture as German.
[assembly:AssemblyCultureAttribute("de")]
```

Informational attributes

You can use informational attributes to provide additional company or product information for an assembly. The following table describes the informational attributes you can apply to an assembly.

Expand table

Informational attribute	Description
AssemblyCompanyAttribute	String value specifying a company name.
AssemblyCopyrightAttribute	String value specifying copyright information.
AssemblyFileVersionAttribute	String value specifying the Win32 file version number. This normally defaults to the assembly version.
AssemblyInformationalVersionAttribute	String value specifying version information that is not used by the common language runtime, such as a full product version number. **Note:** If this attribute is applied to an assembly, the string it specifies can be obtained at run time by using the Application.ProductVersion property. The string is also used in the path and registry key provided by the

Informational attribute	Description
	Application.UserAppDataPath and Application.UserAppDataRegistry properties.
AssemblyProductAttribute	String value specifying product information.
AssemblyTrademarkAttribute	String value specifying trademark information.

These attributes can appear on the Windows Properties page of the assembly, or they can be overridden using the **/win32res** compiler option to specify your Win32 resource file.

Assembly manifest attributes

You can use assembly manifest attributes to provide information in the assembly manifest, including title, description, the default alias, and configuration. The following table describes the assembly manifest attributes.

Expand table

Assembly manifest attribute	Description
AssemblyConfigurationAttribute	String value indicating the configuration of the assembly, such as Retail or Debug. The runtime does not use this value.
AssemblyDefaultAliasAttribute	String value specifying a default alias to be used by referencing assemblies. This value provides a friendly name when the name of the assembly itself is not friendly (such as a GUID value). This value can also be used as a short form of the full assembly name.
AssemblyDescriptionAttribute	String value specifying a short description that summarizes the nature and purpose of the assembly.
AssemblyTitleAttribute	String value specifying a friendly name for the assembly. For example, an assembly named *comdlg* might have the title Microsoft Common Dialog Control.

Strong name attributes

You can use strong name attributes to set a strong name for an assembly. The following table describes the strong name attributes.

Expand table

Strong name attribute	Description
AssemblyDelaySignAttribute	Boolean value indicating that delay signing is being used.
AssemblyKeyFileAttribute	String value indicating the name of the file that contains either the public key (if using delay signing) or both the public and private keys passed as a parameter to the constructor of this attribute. Note that the file name is relative to the output file path (the *.exe* or *.dll*), not the source file path.
AssemblyKeyNameAttribute	Indicates the key container that contains the key pair passed as a parameter to the constructor of this attribute.

The following code example shows the attributes to apply when using delay signing to create a strong-named assembly with a public key file called *myKey.snk*.

C#

```
[assembly:AssemblyKeyFileAttribute("myKey.snk")]
[assembly:AssemblyDelaySignAttribute(true)]
```

See also

- Create assemblies
- Assembly attribute MSBuild properties

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Set assembly attributes in a project file

Article • 01/25/2024

You can use an MSBuild property to transform package-related project properties into assembly attributes in a generated code file. Further, you can use MSBuild items to add arbitrary assembly attributes to the generated file.

Use package properties as assembly attributes

The GenerateAssemblyInfo MSBuild property controls `AssemblyInfo` attribute generation for a project. When the `GenerateAssemblyInfo` value is `true` (which is the default), package-related project properties are transformed into assembly attributes. The following table lists the project properties that generate the attributes. It also lists the properties that you can use to disable that generation on a per-attribute basis, for example:

XML

```
<PropertyGroup>

<GenerateNeutralResourcesLanguageAttribute>false</GenerateNeutralResourcesLanguageA
ttribute>
</PropertyGroup>
```

Expand table

MSBuild property	Assembly attribute	Property to disable attribute generation
`Company`	AssemblyCompanyAttribute	`GenerateAssemblyCompanyAttribute`
`Configuration`	AssemblyConfigurationAttribute	`GenerateAssemblyConfigurationAttribute`
`Copyright`	AssemblyCopyrightAttribute	`GenerateAssemblyCopyrightAttribute`
`Description`	AssemblyDescriptionAttribute	`GenerateAssemblyDescriptionAttribute`
`FileVersion`	AssemblyFileVersionAttribute	`GenerateAssemblyFileVersionAttribute`
`InformationalVersion`	AssemblyInformationalVersionAttribute	`GenerateAssemblyInformationalVersionAttribute`
`Product`	AssemblyProductAttribute	`GenerateAssemblyProductAttribute`
`AssemblyTitle`	AssemblyTitleAttribute	`GenerateAssemblyTitleAttribute`
`AssemblyVersion`	AssemblyVersionAttribute	`GenerateAssemblyVersionAttribute`
`NeutralLanguage`	NeutralResourcesLanguageAttribute	`GenerateNeutralResourcesLanguageAttribute`

Notes about these settings:

- `AssemblyVersion` and `FileVersion` default to the value of `$(Version)` without the suffix. For example, if `$(Version)` is `1.2.3-beta.4`, then the value would be `1.2.3`.
- `InformationalVersion` defaults to the value of `$(Version)`.
- If the `$(SourceRevisionId)` property is present, it's appended to `InformationalVersion`. You can disable this behavior using `IncludeSourceRevisionInInformationalVersion`.
- `Copyright` and `Description` properties are also used for NuGet metadata.
- `Configuration`, which defaults to `Debug`, is shared with all MSBuild targets. You can set it via the `--configuration` option of `dotnet` commands, for example, dotnet pack.
- Some of the properties are used when creating a NuGet package. For more information, see Package properties.

Set arbitrary attributes

It's possible to add your own assembly attributes to the generated file as well. To do so, define `<AssemblyAttribute>` MSBuild items that tell the SDK what type of attribute to create. These items should also include any constructor parameters that are required for that attribute. For example, the System.Reflection.AssemblyMetadataAttribute attribute has a constructor that takes two strings:

- A name to describe an arbitrary value.
- The value to store.

If you had a `Date` property in MSBuild that contained the date when an assembly was created, you could use `AssemblyMetadataAttribute` to embed that date into the assembly attributes using the following MSBuild code:

XML

```xml
<ItemGroup>
  <!-- Include must be the fully qualified .NET type name of the Attribute to
create. -->
  <AssemblyAttribute Include="System.Reflection.AssemblyMetadataAttribute">
    <!-- _Parameter1, _Parameter2, etc. correspond to the
        matching parameter of a constructor of that .NET attribute type -->
    <_Parameter1>BuildDate</_Parameter1>
    <_Parameter2>$(Date)</_Parameter2>
  </AssemblyAttribute>
</ItemGroup>
```

This item tells the .NET SDK to emit the following C# (or equivalent F# or Visual Basic) as an assembly-level attribute:

C#

```
[assembly: System.Reflection.AssemblyMetadataAttribute("BuildDate", "01/19/2024")]
```

(The actual date string would be whatever you provided at the time of the build.)

Migrate from .NET Framework

If you migrate your .NET Framework project to .NET 6 or later, you might encounter an error related to duplicate assembly info files. That's because .NET Framework project templates create a code file with assembly info attributes set. The file is typically located at *.\Properties\AssemblyInfo.cs* or *.\Properties\AssemblyInfo.vb*. However, SDK-style projects also *generate* this file for you based on the project settings.

When porting your code to .NET 6 or later, do one of the following:

- Disable the generation of the temporary code file that contains the assembly info attributes by setting `GenerateAssemblyInfo` to `false` in your project file. This enables you to keep your *AssemblyInfo* file.
- Migrate the settings in the *AssemblyInfo* file to the project file, and then delete the *AssemblyInfo* file.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Strong-named assemblies

Article • 09/15/2021

Strong-naming an assembly creates a unique identity for the assembly, and can prevent assembly conflicts.

What makes a strong-named assembly?

A strong named assembly is generated by using the private key that corresponds to the public key distributed with the assembly, and the assembly itself. The assembly includes the assembly manifest, which contains the names and hashes of all the files that make up the assembly. Assemblies that have the same strong name should be identical.

You can strong-name assemblies by using Visual Studio or a command-line tool. For more information, see How to: Sign an assembly with a strong name or Sn.exe (Strong Name tool).

When a strong-named assembly is created, it contains the simple text name of the assembly, the version number, optional culture information, a digital signature, and the public key that corresponds to the private key used for signing.

> ⚠ **Warning**
>
> Do not rely on strong names for security. They provide a unique identity only.

Why strong-name your assemblies?

For .NET Framework, strong-named assemblies are useful in the following scenarios:

- You want to enable your assemblies to be referenced by strong-named assemblies, or you want to give `friend` access to your assemblies from other strong-named assemblies.
- An app needs access to different versions of the same assembly. This means you need different versions of an assembly to load side by side in the same app domain without conflict. For example, if different extensions of an API exist in assemblies that have the same simple name, strong-naming provides a unique identity for each version of the assembly.

- You do not want to negatively affect performance of apps using your assembly, so you want the assembly to be domain neutral. This requires strong-naming because a domain-neutral assembly must be installed in the global assembly cache.
- You want to centralize servicing for your app by applying publisher policy, which means the assembly must be installed in the global assembly cache.

For .NET Core and .NET 5+, strong-named assemblies do not provide material benefits. The runtime never validates the strong-name signature, nor does it use the strong-name for assembly binding.

If you are an open-source developer and you want the identity benefits of a strong-named assembly for better compatibility with .NET Framework, consider checking in the private key associated with an assembly to your source control system.

See also

- Global assembly cache
- How to: Sign an assembly with a strong name
- Sn.exe (Strong Name tool)
- Create and use strong-named assemblies

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Create and use strong-named assemblies

Article • 09/15/2021

A strong name consists of the assembly's identity—its simple text name, version number, and culture information (if provided)—plus a public key and a digital signature. It is generated from an assembly file using the corresponding private key. (The assembly file contains the assembly manifest, which contains the names and hashes of all the files that make up the assembly.)

> ⚠ **Warning**
>
> Do not rely on strong names for security. They provide a unique identity only.

A strong-named assembly can only use types from other strong-named assemblies. Otherwise, the integrity of the strong-named assembly would be compromised.

> ⓘ **Note**
>
> Although .NET Core supports strong-named assemblies, and all assemblies in the .NET Core library are signed, the majority of third-party assemblies do not need strong names. For more information, see **Strong Name Signing** on GitHub.

Strong name scenario

The following scenario outlines the process of signing an assembly with a strong name and later referencing it by that name.

1. Assembly A is created with a strong name using one of the following methods:

    - Using a development environment that supports creating strong names, such as Visual Studio.

    - Creating a cryptographic key pair using the Strong Name tool (Sn.exe) and assigning that key pair to the assembly using either a command-line compiler or the Assembly Linker (Al.exe). The Windows SDK provides both Sn.exe and Al.exe.

2. The development environment or tool signs the hash of the file containing the assembly's manifest with the developer's private key. This digital signature is stored in the portable executable (PE) file that contains Assembly A's manifest.

3. Assembly B is a consumer of Assembly A. The reference section of Assembly B's manifest includes a token that represents Assembly A's public key. A token is a portion of the full public key and is used rather than the key itself to save space.

4. The common language runtime verifies the strong name signature when the assembly is placed in the global assembly cache. When binding by strong name at run time, the common language runtime compares the key stored in Assembly B's manifest with the key used to generate the strong name for Assembly A. If the .NET security checks pass and the bind succeeds, Assembly B has a guarantee that Assembly A's bits have not been tampered with and that these bits actually come from the developers of Assembly A.

> **Note**
>
> This scenario doesn't address trust issues. Assemblies can carry full Microsoft Authenticode signatures in addition to a strong name. Authenticode signatures include a certificate that establishes trust. It's important to note that strong names don't require code to be signed in this way. Strong names only provide a unique identity.

Bypass signature verification of trusted assemblies

Starting with the .NET Framework 3.5 Service Pack 1, strong-name signatures are not validated when an assembly is loaded into a full-trust application domain, such as the default application domain for the `MyComputer` zone. This is referred to as the strong-name bypass feature. In a full-trust environment, demands for StrongNameIdentityPermission always succeed for signed, full-trust assemblies, regardless of their signature. The strong-name bypass feature avoids the unnecessary overhead of strong-name signature verification of full-trust assemblies in this situation, allowing the assemblies to load faster.

The bypass feature applies to any assembly that is signed with a strong name and that has the following characteristics:

- Fully trusted without StrongName evidence (for example, has `MyComputer` zone evidence).
- Loaded into a fully trusted AppDomain.
- Loaded from a location under the ApplicationBase property of that AppDomain.
- Not delay-signed.

This feature can be disabled for individual applications or for a computer. See How to: Disable the strong-name bypass feature.

Related topics

Title	Description
How to: Create a public-private key pair	Describes how to create a cryptographic key pair for signing an assembly.
How to: Sign an assembly with a strong name	Describes how to create a strong-named assembly.
Enhanced strong naming	Describes enhancements to strong-names in the .NET Framework 4.5.
How to: Reference a strong-named assembly	Describes how to reference types or resources in a strong-named assembly at compile time or run time.
How to: Disable the strong-name bypass feature	Describes how to disable the feature that bypasses the validation of strong-name signatures. This feature can be disabled for all or for specific applications.
Create assemblies	Provides an overview of single-file and multifile assemblies.
How to delay sign an assembly in Visual Studio	Explains how to sign an assembly with a strong name after the assembly has been created.
Sn.exe (Strong Name tool)	Describes the tool included in the .NET Framework that helps create assemblies with strong names. This tool provides options for key management, signature generation, and signature verification.
Al.exe (Assembly linker)	Describes the tool included in the .NET Framework that generates a file that has an assembly manifest from modules or resource files.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to: Create a public-private key pair

Article • 09/15/2021

To sign an assembly with a strong name, you must have a public/private key pair. This public and private cryptographic key pair is used during compilation to create a strong-named assembly. You can create a key pair using the Strong Name tool (Sn.exe). Key pair files usually have an *.snk* extension.

> **Note**
>
> In Visual Studio, the C# and Visual Basic project property pages include a **Signing** tab that enables you to select existing key files or to generate new key files without using *Sn.exe*. In Visual C++, you can specify the location of an existing key file in the **Advanced** property page in the **Linker** section of the **Configuration Properties** section of the **Property Pages** window. The use of the **AssemblyKeyFileAttribute** attribute to identify key file pairs was made obsolete beginning with Visual Studio 2005.

Create a key pair

To create a key pair, at a command prompt, type the following command:

sn –k *<file name>*

In this command, *file name* is the name of the output file containing the key pair.

The following example creates a key pair called *sgKey.snk*.

Windows Command Prompt

```
sn -k sgKey.snk
```

If you intend to delay sign an assembly and you control the whole key pair (which is unlikely outside test scenarios), you can use the following commands to generate a key pair and then extract the public key from it into a separate file. First, create the key pair:

Windows Command Prompt

```
sn -k keypair.snk
```

Next, extract the public key from the key pair and copy it to a separate file:

Windows Command Prompt

```
sn -p keypair.snk public.snk
```

Once you create the key pair, you must put the file where the strong name signing tools can find it.

When signing an assembly with a strong name, the Assembly Linker (Al.exe) looks for the key file relative to the current directory and to the output directory. When using command-line compilers, you can simply copy the key to the current directory containing your code modules.

If you are using an earlier version of Visual Studio that does not have a **Signing** tab in the project properties, the recommended key file location is the project directory with the file attribute specified as follows:

C#

```
[assembly:AssemblyKeyFileAttribute("keyfile.snk")]
```

See also

- Create and use strong-named assemblies

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to: Sign an assembly with a strong name

Article • 08/31/2022

> **Note**
>
> Although .NET Core supports strong-named assemblies, and all assemblies in the .NET Core library are signed, the majority of third-party assemblies do not need strong names. For more information, see **Strong Name Signing** on GitHub.

There are a number of ways to sign an assembly with a strong name:

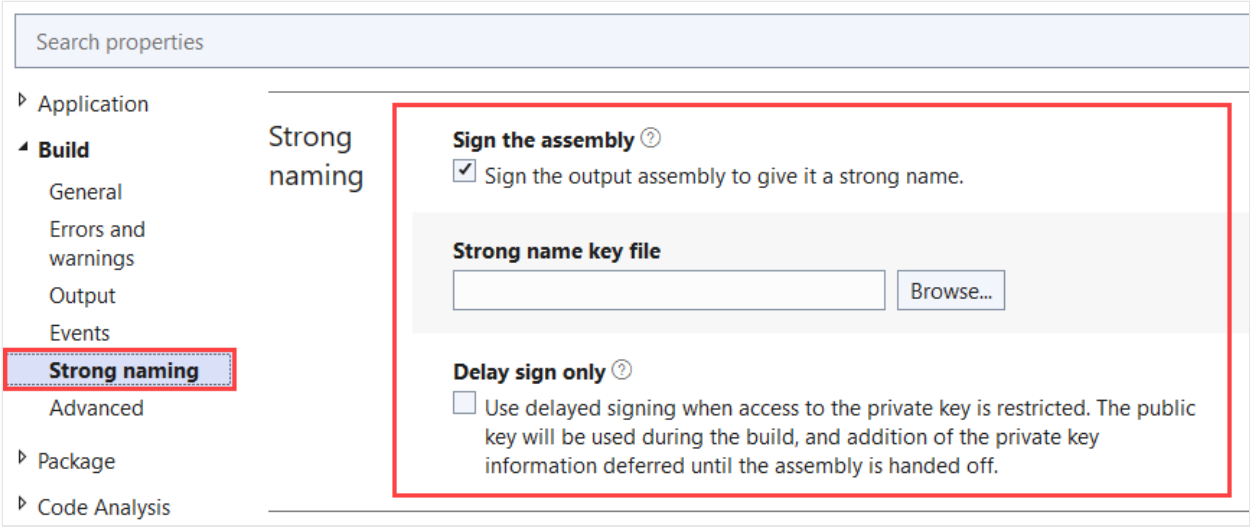
- By using the **Build** > **Strong naming** page in the project designer for a project in Visual Studio. This is the easiest and most convenient way to sign an assembly with a strong name.
- By using the Assembly Linker (Al.exe) to link a .NET Framework code module (a *.netmodule* file) with a key file.
- By using assembly attributes to insert the strong name information into your code. You can use either the AssemblyKeyFileAttribute or the AssemblyKeyNameAttribute attribute, depending on where the key file to be used is located.
- By using compiler options.

You must have a cryptographic key pair to sign an assembly with a strong name. For more information about creating a key pair, see How to: Create a public-private key pair.

Create and sign an assembly with a strong name by using Visual Studio

1. In **Solution Explorer**, open the shortcut menu for the project, and then choose **Properties**.
2. Under the **Build** tab you'll find a **Strong naming** node.
3. Select the **Sign the assembly** checkbox, which expands the options.
4. Select the **Browse** button to choose a **Strong name key file** path.

> **Note**
>
> In order to **delay sign an assembly**, choose a public key file.



Create and sign an assembly with a strong name by using the Assembly Linker

Open Visual Studio Developer Command Prompt or Visual Studio Developer PowerShell, and enter the following command:

al /out:<*assemblyName*> <*moduleName*> **/keyfile:**<*keyfileName*>

Where:

- *assemblyName* is the name of the strongly signed assembly (a *.dll* or *.exe* file) that Assembly Linker will emit.
- *moduleName* is the name of a .NET Framework code module (a *.netmodule* file) that includes one or more types. You can create a *.netmodule* file by compiling your code with the `/target:module` switch in C# or Visual Basic.
- *keyfileName* is the name of the container or file that contains the key pair. Assembly Linker interprets a relative path in relation to the current directory.

The following example signs the assembly *MyAssembly.dll* with a strong name by using the key file *sgKey.snk*.

Console

```
al /out:MyAssembly.dll MyModule.netmodule /keyfile:sgKey.snk
```

For more information about this tool, see Assembly Linker.

Sign an assembly with a strong name by using attributes

1. Add the System.Reflection.AssemblyKeyFileAttribute or AssemblyKeyNameAttribute attribute to your source code file, and specify the name of the file or container that contains the key pair to use when signing the assembly with a strong name.

2. Compile the source code file normally.

> **Note**
>
> The C# and Visual Basic compilers issue compiler warnings (CS1699 and BC41008, respectively) when they encounter the **AssemblyKeyFileAttribute** or **AssemblyKeyNameAttribute** attribute in source code. You can ignore the warnings.

The following example uses the AssemblyKeyFileAttribute attribute with a key file called *keyfile.snk*, which is located in the directory where the assembly is compiled.

C#

```
[assembly:AssemblyKeyFileAttribute("keyfile.snk")]
```

You can also delay sign an assembly when compiling your source file. For more information, see Delay-sign an assembly.

Sign an assembly with a strong name by using the compiler

Compile your source code file or files with the `/keyfile` or `/delaysign` compiler option in C# and Visual Basic, or the `/KEYFILE` or `/DELAYSIGN` linker option in C++. After the option name, add a colon and the name of the key file. When using command-line compilers, you can copy the key file to the directory that contains your source code files.

For information on delay signing, see Delay-sign an assembly.

The following example uses the C# compiler and signs the assembly *UtilityLibrary.dll* with a strong name by using the key file *sgKey.snk*.

Windows Command Prompt

```
csc /t:library UtilityLibrary.cs /keyfile:sgKey.snk
```

See also

- Create and use strong-named assemblies
- How to: Create a public-private key pair
- Al.exe (Assembly Linker)
- Delay-sign an assembly
- Strong-name APIs throw PlatformNotSupportedException
- Manage assembly and manifest signing
- Signing page, Project Designer

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Enhanced strong naming

Article • 08/23/2022

A strong name signature is an identity mechanism in the .NET Framework for identifying assemblies. It is a public-key digital signature that is typically used to verify the integrity of data being passed from an originator (signer) to a recipient (verifier). This signature is used as a unique identity for an assembly and ensures that references to the assembly are not ambiguous. The assembly is signed as part of the build process and then verified when it is loaded.

Strong name signatures help prevent malicious parties from tampering with an assembly and then re-signing the assembly with the original signer's key. However, strong name keys don't contain any reliable information about the publisher, nor do they contain a certificate hierarchy. A strong name signature does not guarantee the trustworthiness of the person who signed the assembly or indicate whether that person was a legitimate owner of the key; it indicates only that the owner of the key signed the assembly. Therefore, we do not recommend using a strong name signature as a security validator for trusting third-party code. Microsoft Authenticode is the recommended way to authenticate code.

Limitations of conventional strong names

The strong naming technology used in versions before the .NET Framework 4.5 has the following shortcomings:

- Keys are constantly under attack, and improved techniques and hardware make it easier to infer a private key from a public key. To guard against attacks, larger keys are necessary. .NET Framework versions before the .NET Framework 4.5 provide the ability to sign with any size key (the default size is 1024 bits), but signing an assembly with a new key breaks all binaries that reference the older identity of the assembly. Therefore, it is extremely difficult to upgrade the size of a signing key if you want to maintain compatibility.

- Strong name signing supports only the SHA-1 algorithm. SHA-1 has recently been found to be inadequate for secure hashing applications. Therefore, a stronger algorithm (SHA-256 or greater) is necessary. It is possible that SHA-1 will lose its FIPS-compliant standing, which would present problems for those who choose to use only FIPS-compliant software and algorithms.

Advantages of enhanced strong names

The main advantages of enhanced strong names are compatibility with pre-existing strong names and the ability to claim that one identity is equivalent to another:

- Developers who have pre-existing signed assemblies can migrate their identities to the SHA-2 algorithms while maintaining compatibility with assemblies that reference the old identities.
- Developers who create new assemblies and are not concerned with pre-existing strong name signatures can use the more secure SHA-2 algorithms and sign the assemblies as they always have.

Use enhanced strong names

Strong name keys consist of a signature key and an identity key. The assembly is signed with the signature key and is identified by the identity key. Prior to .NET Framework 4.5, these two keys were identical. Starting with .NET Framework 4.5, the identity key remains the same as in earlier .NET Framework versions, but the signature key is enhanced with a stronger hash algorithm. In addition, the signature key is signed with the identity key to create a counter-signature.

The AssemblySignatureKeyAttribute attribute enables the assembly metadata to use the pre-existing public key for assembly identity, which allows old assembly references to continue to work. The AssemblySignatureKeyAttribute attribute uses the counter-signature to ensure that the owner of the new signature key is also the owner of the old identity key.

Sign with SHA-2, without key migration

Run the following commands from a command prompt to sign an assembly without migrating a strong name signature:

1. Generate the new identity key (if necessary).

   Console

   ```
   sn -k IdentityKey.snk
   ```

2. Extract the identity public key, and specify that a SHA-2 algorithm should be used when signing with this key.

   Console

```
sn -p IdentityKey.snk IdentityPubKey.snk sha256
```

3. Delay-sign the assembly with the identity public key file.

```
Console

csc MyAssembly.cs /keyfile:IdentityPubKey.snk /delaySign+
```

4. Re-sign the assembly with the full identity key pair.

```
Console

sn -Ra MyAssembly.exe IdentityKey.snk
```

Sign with SHA-2, with key migration

Run the following commands from a command prompt to sign an assembly with a migrated strong name signature.

1. Generate an identity and signature key pair (if necessary).

```
Console

sn -k IdentityKey.snk
sn -k SignatureKey.snk
```

2. Extract the signature public key, and specify that a SHA-2 algorithm should be used when signing with this key.

```
Console

sn -p SignatureKey.snk SignaturePubKey.snk sha256
```

3. Extract the identity public key, which determines the hash algorithm that generates a counter-signature.

```
Console

sn -p IdentityKey.snk IdentityPubKey.snk
```

4. Generate the parameters for an AssemblySignatureKeyAttribute attribute, and attach the attribute to the assembly.

Console

```
sn -a IdentityPubKey.snk IdentityKey.snk SignaturePubKey.snk
```

This produces output similar to the following.

Output

```
Information for key migration attribute.
(System.Reflection.AssemblySignatureKeyAttribute):
publicKey=
002400000c800000940000000602000000240000525341310004000001000100005a3a81
ac0a519
d96244a9c589fc147c7d403e40ccf184fc290bdd06c7339389a76b738e255a2bce1d56c
3e7e936
e4fc87d45adc82ca94c716b50a65d39d373eea033919a613e4341c66863cb2dc622bcb5
41762b4
3893434d219d1c43f07e9c83fada2aed400b9f6e44ff05e3ecde6c2827830b8f43f7ac8
e3270a3
4d153cdd

counterSignature=
e3cf7c211678c4d1a7b8fb20276c894ab74c29f0b5a34de4d61e63d4a997222f78cdcbf
e4c91eb
e1ddf9f3505a32edcb2a76f34df0450c4f61e376b70fa3cdeb7374b1b8e2078b121e2ee
6e8c6a8
ed661cc35621b4af53ac29c9e41738f199a81240e8fd478c887d1a30729d34e954a97cd
dce66e3
ae5fec2c682e57b7442738
```

This output can then be transformed into an AssemblySignatureKeyAttribute.

C#

```
[assembly:System.Reflection.AssemblySignatureKeyAttribute(
"002400000c800000940000000602000000240000525341310004000001000100005a3a8
1ac0a519d96244a9c589fc147c7d403e40ccf184fc290bdd06c7339389a76b738e255a2
bce1d56c3e7e936e4fc87d45adc82ca94c716b50a65d39d373eea033919a613e4341c66
863cb2dc622bcb541762b43893434d219d1c43f07e9c83fada2aed400b9f6e44ff05e3e
cde6c2827830b8f43f7ac8e3270a34d153cdd",
"e3cf7c211678c4d1a7b8fb20276c894ab74c29f0b5a34de4d61e63d4a997222f78cdcb
fe4c91ebe1ddf9f3505a32edcb2a76f34df0450c4f61e376b70fa3cdeb7374b1b8e2078
b121e2ee6e8c6a8ed661cc35621b4af53ac29c9e41738f199a81240e8fd478c887d1a30
729d34e954a97cddce66e3ae5fec2c682e57b7442738"
)]
```

5. Delay-sign the assembly with the identity public key.

Console

```
csc MyAssembly.cs /keyfile:IdentityPubKey.snk /delaySign+
```

6. Fully sign the assembly with the signature key pair.

```Console
sn -Ra MyAssembly.exe SignatureKey.snk
```

See also

- Create and use strong-named assemblies

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to: Reference a strong-named assembly

Article • 09/15/2021

The process for referencing types or resources in a strong-named assembly is usually transparent. You can make the reference either at compile time (early binding) or at run time.

A compile-time reference occurs when you indicate to the compiler that the assembly to be compiled explicitly references another assembly. When you use compile-time referencing, the compiler automatically gets the public key of the targeted strong-named assembly and places it in the assembly reference of the assembly being compiled.

> **Note**
>
> A strong-named assembly can only use types from other strong-named assemblies. Otherwise, the security of the strong-named assembly would be compromised.

Make a compile-time reference to a strong-named assembly

At a command prompt, type the following command:

<compiler command> **/reference:***<assembly name>*

In this command, *compiler command* is the compiler command for the language you are using, and *assembly name* is the name of the strong-named assembly being referenced. You can also use other compiler options, such as the **/t:library** option for creating a library assembly.

The following example creates an assembly called *myAssembly.dll* that references a strong-named assembly called *myLibAssembly.dll* from a code module called *myAssembly.cs*.

Windows Command Prompt

```
csc /t:library myAssembly.cs /reference:myLibAssembly.dll
```

Make a run-time reference to a strong-named assembly

When you make a run-time reference to a strong-named assembly, for example by using the Assembly.Load or Assembly.GetType method, you must use the display name of the referenced strong-named assembly. The syntax of a display name is as follows:

<assembly name>, *<version number>*, *<culture>*, *<public key token>*

For example:

```
Console
```

```
myDll, Version=1.1.0.0, Culture=en, PublicKeyToken=03689116d3a4ae33
```

In this example, `PublicKeyToken` is the hexadecimal form of the public key token. If there is no culture value, use `Culture=neutral`.

The following code example shows how to use this information with the Assembly.Load method.

```
C#
```

```
Assembly myDll =
    Assembly.Load("myDll, Version=1.0.0.1, Culture=neutral,
PublicKeyToken=9b35aa32c18d4fb1");
```

You can print the hexadecimal format of the public key and public key token for a specific assembly by using the following Strong Name (Sn.exe) command:

sn -Tp < *assembly* >

If you have a public key file, you can use the following command instead (note the difference in case on the command-line option):

sn -tp < *public key file* >

See also

- Create and use strong-named assemblies

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to: Disable the strong-name bypass feature

Article • 09/15/2021

Starting with the .NET Framework version 3.5 Service Pack 1 (SP1), strong-name signatures are not validated when an assembly is loaded into a full-trust AppDomain object, such as the default AppDomain for the `MyComputer` zone. This is referred to as the strong-name bypass feature. In a full-trust environment, demands for StrongNameIdentityPermission always succeed for signed, full-trust assemblies regardless of their signature. The only restriction is that the assembly must be fully trusted because its zone is fully trusted. Because the strong name is not a determining factor under these conditions, there is no reason for it to be validated. Bypassing the validation of strong-name signatures provides significant performance improvements.

The bypass feature applies to any full-trust assembly that is not delay-signed and that is loaded into any full-trust AppDomain from the directory specified by its ApplicationBase property.

You can override the bypass feature for all applications on a computer by setting a registry key value. You can override the setting for a single application by using an application configuration file. You cannot reinstate the bypass feature for a single application if it has been disabled by the registry key.

When you override the bypass feature, the strong name is validated only for correctness; it is not checked for a StrongNameIdentityPermission. If you want to confirm a specific strong name, you have to perform that check separately.

> **ⓘ Important**
>
> The ability to force strong-name validation depends on a registry key, as described in the following procedure. If an application is running under an account that does not have access control list (ACL) permission to access that registry key, the setting is ineffective. You must ensure that ACL rights are configured for this key so that it can be read for all assemblies.

Disable the strong-name bypass feature for all applications

- On 32-bit computers, in the system registry, create a DWORD entry with a value of 0 named `AllowStrongNameBypass` under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework key.
- On 64-bit computers, in the system registry, create a DWORD entry with a value of 0 named `AllowStrongNameBypass` under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework and HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\.NETFramework keys.

Disable the strong-name bypass feature for a single application

1. Open or create the application configuration file.

   For more information about this file, see the Application Configuration Files section in Configure apps.

2. Add the following entry:

```xml
<configuration>
  <runtime>
    <bypassTrustedAppStrongNames enabled="false" />
  </runtime>
</configuration>
```

You can restore the bypass feature for the application by removing the configuration file setting or by setting the attribute to `true`.

> **Note**
>
> You can turn strong-name validation on and off for an application only if the bypass feature is enabled for the computer. If the bypass feature has been turned off for the computer, strong names are validated for all applications and you cannot bypass validation for a single application.

See also

- Sn.exe (Strong Name Tool)

- <bypassTrustedAppStrongNames> element
- Create and use strong-named assemblies

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Delay-sign an assembly

Article • 04/29/2022

An organization can have a closely guarded key pair that developers can't access on a daily basis. The public key is often available, but access to the private key is restricted to only a few individuals. When developing assemblies with strong names, each assembly that references the strong-named target assembly contains the token of the public key used to give the target assembly a strong name. This requires that the public key be available during the development process.

You can use delayed or partial signing at build time to reserve space in the portable executable (PE) file for the strong name signature, but defer the actual signing until some later stage, usually just before shipping the assembly.

To delay-sign an assembly:

1. Get the public key portion of the key pair from the organization that will do the eventual signing. Typically this key is in the form of an *.snk* file, which can be created using the Strong Name tool (Sn.exe) provided by the Windows SDK.

2. Annotate the source code for the assembly with two custom attributes from System.Reflection:

   - AssemblyKeyFileAttribute, which passes the name of the file containing the public key as a parameter to its constructor.

   - AssemblyDelaySignAttribute, which indicates that delay signing is being used by passing **true** as a parameter to its constructor.

   For example:

   C#

   ```
   [assembly:AssemblyKeyFileAttribute("myKey.snk")]
   [assembly:AssemblyDelaySignAttribute(true)]
   ```

3. The compiler inserts the public key into the assembly manifest and reserves space in the PE file for the full strong name signature. The real public key must be stored while the assembly is built so that other assemblies that reference this assembly can obtain the key to store in their own assembly reference.

4. Because the assembly does not have a valid strong name signature, the verification of that signature must be turned off. You can do this by using the **-Vr** option with

the Strong Name tool.

The following example turns off verification for an assembly called *myAssembly.dll*.

Console

```
sn -Vr myAssembly.dll
```

To turn off verification on platforms where you can't run the Strong Name tool, such as Advanced RISC Machine (ARM) microprocessors, use the **-Vk** option to create a registry file. Import the registry file into the registry on the computer where you want to turn off verification. The following example creates a registry file for `myAssembly.dll`.

Console

```
sn -Vk myRegFile.reg myAssembly.dll
```

With either the **-Vr** or **-Vk** option, you can optionally include an *.snk* file for test key signing.

> ⚠ **Warning**
>
> Do not rely on strong names for security. They provide a unique identity only.

> ⓘ **Note**
>
> If you use delay signing during development with Visual Studio on a 64-bit computer, and you compile an assembly for **Any CPU**, you might have to apply the **-Vr** option twice. (In Visual Studio, **Any CPU** is a value of the **Platform Target** build property; when you compile from the command line, it is the default.) To run your application from the command line or from File Explorer, use the 64-bit version of the **Sn.exe (Strong Name tool)** to apply the **-Vr** option to the assembly. To load the assembly into Visual Studio at design time (for example, if the assembly contains components that are used by other assemblies in your application), use the 32-bit version of the strong-name tool. This is because the just-in-time (JIT) compiler compiles the assembly to 64-bit native code when the assembly is run from the command line, and to 32-bit native code when the assembly is loaded into the design-time environment.

5. Later, usually just before shipping, you submit the assembly to your organization's signing authority for the actual strong name signing using the **-R** option with the Strong Name tool.

   The following example signs an assembly called *myAssembly.dll* with a strong name using the *sgKey.snk* key pair.

   Console

   ```
   sn -R myAssembly.dll sgKey.snk
   ```

See also

- Create assemblies
- How to: Create a public-private key pair
- Sn.exe (Strong Name tool)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to: View assembly contents

Article • 01/12/2022

You can use the Ildasm.exe (IL Disassembler) to view Microsoft intermediate language (MSIL) information in a file. If the file being examined is an assembly, this information can include the assembly's attributes and references to other modules and assemblies. This information can be helpful in determining whether a file is an assembly or part of an assembly and whether the file has references to other modules or assemblies.

To display the contents of an assembly using *Ildasm.exe*, enter **ildasm <assembly name>** at a command prompt. For example, the following command disassembles the *Hello.exe* assembly.

Windows Command Prompt

```
ildasm Hello.exe
```

To view assembly manifest information, double-click the **Manifest** icon in the MSIL Disassembler window.

Example

The following example starts with a basic "Hello World" program. After compiling the program, use *Ildasm.exe* to disassemble the *Hello.exe* assembly and view the assembly manifest.

C#

```
using System;

class MainApp
{
    public static void Main()
    {
        Console.WriteLine("Hello World using C#!");
    }
}
```

Running the command *ildasm.exe* on the *Hello.exe* assembly and double-clicking the **Manifest** icon in the MSIL Disassembler window produces the following output:

Output

```
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         //
.z\V.4..
  .ver 4:0:0:0
}
.assembly Hello
{
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.
ctor(int32) = ( 01 00 08 00 00 00 00 00 )
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ct
or() = ( 01 00 01 00 54 02 16 57 72 61 70 4E 6F 6E 45 78   //
....T..WrapNonEx

63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module Hello.exe
// MVID: {7C2770DB-1594-438D-BAE5-98764C39CCCA}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x00600000
```

The following table describes each directive in the assembly manifest of the *Hello.exe* assembly used in the example:

Expand table

Directive	Description
.assembly extern <assembly name>	Specifies another assembly that contains items referenced by the current module (in this example, `mscorlib`).
.publickeytoken <token>	Specifies the token of the actual key of the referenced assembly.
.ver <version number>	Specifies the version number of the referenced assembly.
.assembly <assembly name>	Specifies the assembly name.
.hash algorithm <int32 value>	Specifies the hash algorithm used.

Directive	Description
.ver <version number>	Specifies the version number of the assembly.
.module <file name>	Specifies the name of the modules that make up the assembly. In this example, the assembly consists of only one file.
.subsystem <value>	Specifies the application environment required for the program. In this example, the value 3 indicates that this executable is run from a console.
.corflags	Currently a reserved field in the metadata.

An assembly manifest can contain a number of different directives, depending on the contents of the assembly. For an extensive list of the directives in the assembly manifest, see the Ecma documentation, especially "Partition II: Metadata Definition and Semantics" and "Partition III: CIL Instruction Set":

- ECMA C# and Common Language Infrastructure standards
- Standard ECMA-335 - Common Language Infrastructure (CLI)

See also

- Application domains and assemblies
- Ildasm.exe (IL Disassembler)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Type forwarding in the common language runtime

Article • 10/14/2022

Type forwarding allows you to move a type to another assembly without having to recompile applications that use the original assembly.

For example, suppose an application uses the `Example` class in an assembly named *Utility.dll*. The developers of *Utility.dll* might decide to refactor the assembly, and in the process they might move the `Example` class to another assembly. If the old version of *Utility.dll* is replaced by the new version of *Utility.dll* and its companion assembly, the application that uses the `Example` class fails because it cannot locate the `Example` class in the new version of *Utility.dll*.

The developers of *Utility.dll* can avoid this by forwarding requests for the `Example` class, using the TypeForwardedToAttribute attribute. If the attribute has been applied to the new version of *Utility.dll*, requests for the `Example` class are forwarded to the assembly that now contains the class. The existing application continues to function normally, without recompilation.

Forward a type

There are four steps to forwarding a type:

1. Move the source code for the type from the original assembly to the destination assembly.

2. In the assembly where the type used to be located, add a TypeForwardedToAttribute for the type that was moved. The following code shows the attribute for a type named `Example` that was moved.

   C#

   ```
   [assembly:TypeForwardedToAttribute(typeof(Example))]
   ```

3. Compile the assembly that now contains the type.

4. Recompile the assembly where the type used to be located, with a reference to the assembly that now contains the type. For example, if you are compiling a C# file from the command line, use the **References** (C# compiler options) option to

specify the assembly that contains the type. In C++, use the #using directive in the source file to specify the assembly that contains the type.

C# type forwarding example

Continuing from the contrived example description above, imagine you're developing the *Utility.dll*, and you have an `Example` class. The *Utility.csproj* is a basic class library:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsing>true</ImplicitUsing>
  </PropertyGroup>

</Project>
```

The `Example` class provides a few properties and overrides Object.ToString:

C#

```csharp
using System;

namespace Common.Objects;

public class Example
{
    public string Message { get; init; } = "Hi friends!";

    public Guid Id { get; init; } = Guid.NewGuid();

    public DateOnly Date { get; init; } =
DateOnly.FromDateTime(DateTime.Today);

    public sealed override string ToString() =>
        $"[{Id} - {Date}]: {Message}";
}
```

Now, imagine that there is a consuming project and it's represented in the *Consumer* assembly. This consuming project references the *Utility* assembly. As an example, it instantiates the `Example` object and writes it to the console in its *Program.cs* file:

C#

```
using System;
using Common.Objects;

Example example = new();

Console.WriteLine(example);
```

When the consuming app runs, it will output the state of the `Example` object. At this point, there is no type forwarding as the *Consuming.csproj* references the *Utility.csproj*. However, the developer's of the *Utility* assembly decide to remove the `Example` object as part of a refactoring. This type is moved to a newly created *Common.csproj*.

By removing this type from the *Utility* assembly, the developers are introducing a breaking change. All consuming projects will break when they update to the latest *Utility* assembly.

Instead of requiring the consuming projects to add a new reference to the *Common* assembly, you can forward the type. Since this type was removed from the *Utility* assembly, you'll need to have the *Utility.csproj* reference the *Common.csproj*:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsing>true</ImplicitUsing>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\Common\Common.csproj" />
  </ItemGroup>

</Project>
```

The preceding C# project now references the newly created *Common* assembly. This could be either a `PackageReference` or a `ProjectReference`. The *Utility* assembly needs to provide the type forwarding information. By convention type forward declarations are usually encapsulated in a single file named `TypeForwarders`, consider the following *TypeForwarders.cs* C# file in the *Utility* assembly:

C#

```
using System.Runtime.CompilerServices;
using Common.Objects;
```

```
[assembly:TypeForwardedTo(typeof(Example))]
```

The *Utility* assembly references the *Common* assembly, and it forwards the `Example` type. If you're to compile the *Utility* assembly with the type forwarding declarations and drop the *Utility.dll* into the *Consuming* bin, the consuming app will work without being compiled.

See also

- TypeForwardedToAttribute
- Type forwarding (C++/CLI)
- #using directive

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Friend assemblies

Article • 09/15/2021

A *friend assembly* is an assembly that can access another assembly's internal (C#) or Friend (Visual Basic) types and members. If you add an assembly attribute to *AssemblyA* to identify *AssemblyB* as a friend assembly, you no longer have to mark types and members in *AssemblyA* as public in order for them to be accessed by *AssemblyB*. This is especially convenient in the following scenarios:

- During unit testing, when test code runs in a separate assembly but requires access to members in the assembly being tested that are marked as `internal` in C# or `Friend` in Visual Basic.
- When you are developing a class library and additions to the library are contained in separate assemblies but require access to members in existing assemblies that are marked as `internal` in C# or `Friend` in Visual Basic.

Remarks

You can use the InternalsVisibleToAttribute attribute to identify one or more friend assemblies for a given assembly. The following example uses the InternalsVisibleToAttribute attribute in *AssemblyA* and specifies assembly *AssemblyB* as a friend assembly. This gives assembly *AssemblyB* access to all types and members in *Assembly A* that are marked as `internal` in C# or `Friend` in Visual Basic.

> **Note**
>
> When you compile an assembly like *AssemblyB* that will access internal types or internal members of another assembly like *AssemblyA*, you must explicitly specify the name of the output file (*.exe* or *.dll*) by using the **-out** compiler option. This is required because the compiler has not yet generated the name for the assembly it is building at the time it is binding to external references. For more information, see **OutputAssembly (C#)** or **-out (Visual Basic)**.

C#

```
using System.Runtime.CompilerServices;
using System;

[assembly: InternalsVisibleTo("AssemblyB")]
```

```
// The class is internal by default.
class FriendClass
{
    public void Test()
    {
        Console.WriteLine("Sample Class");
    }
}

// Public class that has an internal method.
public class ClassWithFriendMethod
{
    internal void Test()
    {
        Console.WriteLine("Sample Method");
    }

}
```

Only assemblies that you explicitly specify as friends can access `internal` (C#) or `Friend` (Visual Basic) types and members. For example, if *AssemblyB* is a friend of *Assembly A* and *Assembly C* references *AssemblyB*, *Assembly C* does not have access to `internal` (C#) or `Friend` (Visual Basic) types in *Assembly A*.

The compiler performs some basic validation of the friend assembly name passed to the InternalsVisibleToAttribute attribute. If *Assembly A* declares *AssemblyB* as a friend assembly, the validation rules are as follows:

- If *Assembly A* is strong named, *AssemblyB* must also be strong named. The friend assembly name that is passed to the attribute must consist of the assembly name and the public key of the strong-name key that is used to sign *AssemblyB*.

  The friend assembly name that is passed to the InternalsVisibleToAttribute attribute cannot be the strong name of *AssemblyB*. Don't include the assembly version, culture, architecture, or public key token.

- If *Assembly A* is not strong named, the friend assembly name should consist of only the assembly name. For more information, see How to: Create unsigned friend assemblies.

- If *AssemblyB* is strong named, you must specify the strong-name key for *AssemblyB* by using the project setting or the command-line `/keyfile` compiler option. For more information, see How to: Create signed friend assemblies.

The StrongNameIdentityPermission class also provides the ability to share types, with the following differences:

- StrongNameIdentityPermission applies to an individual type, while a friend assembly applies to the whole assembly.
- If there are hundreds of types in *Assembly A* that you want to share with *AssemblyB*, you have to add StrongNameIdentityPermission to all of them. If you use a friend assembly, you only need to declare the friend relationship once.
- If you use StrongNameIdentityPermission, the types you want to share have to be declared as public. If you use a friend assembly, the shared types are declared as `internal` (C#) or `Friend` (Visual Basic).

For information about how to access an assembly's `internal` (C#) or `Friend` (Visual Basic) types and methods from a module file (a file with the *.netmodule* extension), see **ModuleAssemblyName** (C#) or -moduleassemblyname (Visual Basic).

See also

- InternalsVisibleToAttribute
- StrongNameIdentityPermission
- How to: Create unsigned friend assemblies
- How to: Create signed friend assemblies
- Assemblies in .NET
- C# programming guide
- Programming concepts (Visual Basic)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to: Create unsigned friend assemblies

Article • 09/15/2021

This example shows how to use friend assemblies with assemblies that are unsigned.

Create an assembly and a friend assembly

1. Open a command prompt.

2. Create a C# or Visual Basic file named *friend_unsigned_A* that contains the following code. The code uses the InternalsVisibleToAttribute attribute to declare *friend_unsigned_B* as a friend assembly.

   C#

   ```
   // friend_unsigned_A.cs
   // Compile with:
   // csc /target:library friend_unsigned_A.cs
   using System.Runtime.CompilerServices;
   using System;

   [assembly: InternalsVisibleTo("friend_unsigned_B")]

   // Type is internal by default.
   class Class1
   {
       public void Test()
       {
           Console.WriteLine("Class1.Test");
       }
   }

   // Public type with internal member.
   public class Class2
   {
       internal void Test()
       {
           Console.WriteLine("Class2.Test");
       }
   }
   ```

3. Compile and sign *friend_unsigned_A* by using the following command:

   C#

```
csc /target:library friend_unsigned_A.cs
```

4. Create a C# or Visual Basic file named *friend_unsigned_B* that contains the following code. Because *friend_unsigned_A* specifies *friend_unsigned_B* as a friend assembly, the code in *friend_unsigned_B* can access `internal` (C#) or `Friend` (Visual Basic) types and members from *friend_unsigned_A*.

```
C#

// friend_unsigned_B.cs
// Compile with:
// csc /r:friend_unsigned_A.dll /out:friend_unsigned_B.exe
friend_unsigned_B.cs
public class Program
{
    static void Main()
    {
        // Access an internal type.
        Class1 inst1 = new Class1();
        inst1.Test();

        Class2 inst2 = new Class2();
        // Access an internal member of a public type.
        inst2.Test();

        System.Console.ReadLine();
    }
}
```

5. Compile *friend_unsigned_B* by using the following command.

```
C#

csc /r:friend_unsigned_A.dll /out:friend_unsigned_B.exe
friend_unsigned_B.cs
```

   The name of the assembly that is generated by the compiler must match the friend assembly name that is passed to the InternalsVisibleToAttribute attribute. You must explicitly specify the name of the output assembly (*.exe* or *.dll*) by using the `-out` compiler option. For more information, see **OutputAssembly** (C# compiler options) or -out (Visual Basic)..

6. Run the *friend_unsigned_B.exe* file.

   The program outputs two strings: **Class1.Test** and **Class2.Test**.

.NET security

There are similarities between the InternalsVisibleToAttribute attribute and the StrongNameIdentityPermission class. The main difference is that StrongNameIdentityPermission can demand security permissions to run a particular section of code, whereas the InternalsVisibleToAttribute attribute controls the visibility of `internal` or `Friend` (Visual Basic) types and members.

See also

- InternalsVisibleToAttribute
- Assemblies in .NET
- Friend assemblies
- How to: Create signed friend assemblies
- C# programming guide
- Programming concepts (Visual Basic)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to: Create signed friend assemblies

Article • 09/15/2021

This example shows how to use friend assemblies with assemblies that have strong names. Both assemblies must be strong named. Although both assemblies in this example use the same keys, you could use different keys for two assemblies.

Create a signed assembly and a friend assembly

1. Open a command prompt.
2. Use the following sequence of commands with the Strong Name tool to generate a keyfile and to display its public key. For more information, see Sn.exe (Strong Name tool).

   a. Generate a strong-name key for this example and store it in the file *FriendAssemblies.snk*:

   ```
   sn -k FriendAssemblies.snk
   ```

   b. Extract the public key from *FriendAssemblies.snk* and put it into *FriendAssemblies.publickey*:

   ```
   sn -p FriendAssemblies.snk FriendAssemblies.publickey
   ```

   c. Display the public key stored in the file *FriendAssemblies.publickey*:

   ```
   sn -tp FriendAssemblies.publickey
   ```

3. Create a C# or Visual Basic file named *friend_signed_A* that contains the following code. The code uses the InternalsVisibleToAttribute attribute to declare *friend_signed_B* as a friend assembly.

   The Strong Name tool generates a new public key every time it runs. Therefore, you must replace the public key in the following code with the public key you just generated, as shown in the following example.

   C#

   ```
   // friend_signed_A.cs
   // Compile with:
   // csc /target:library /keyfile:FriendAssemblies.snk friend_signed_A.cs
   ```

```
using System.Runtime.CompilerServices;

[assembly: InternalsVisibleTo("friend_signed_B,
PublicKey=0024000004800000940000000602000000240000525341310004000001000
100e3aedce99b7e10823920206f8e46cd5558b4ec7345bd1a5b201ffe71660625dcb8f9
a08687d881c8f65a0dcf042f81475d2e88f3e3e273c8311ee40f952db306c02fbfc5d8b
c6ee1e924e6ec8fe8c01932e0648a0d3e5695134af3bb7fab370d3012d083fa6b83179d
d3d031053f72fc1f7da8459140b0af5afc4d2804deccb6")]
class Class1
{
    public void Test()
    {
        System.Console.WriteLine("Class1.Test");
        System.Console.ReadLine();
    }
}
```

4. Compile and sign *friend_signed_A* by using the following command.

```
C#

csc /target:library /keyfile:FriendAssemblies.snk friend_signed_A.cs
```

5. Create a C# or Visual Basic file named *friend_signed_B* that contains the following code. Because *friend_signed_A* specifies *friend_signed_B* as a friend assembly, the code in *friend_signed_B* can access `internal` (C#) or `Friend` (Visual Basic) types and members from *friend_signed_A*. The file contains the following code.

```
C#

// friend_signed_B.cs
// Compile with:
// csc /keyfile:FriendAssemblies.snk /r:friend_signed_A.dll
/out:friend_signed_B.exe friend_signed_B.cs
public class Program
{
    static void Main()
    {
        Class1 inst = new Class1();
        inst.Test();
    }
}
```

6. Compile and sign *friend_signed_B* by using the following command.

```
C#

csc /keyfile:FriendAssemblies.snk /r:friend_signed_A.dll
```

```
/out:friend_signed_B.exe friend_signed_B.cs
```

The name of the assembly generated by the compiler must match the friend assembly name passed to the InternalsVisibleToAttribute attribute. You must explicitly specify the name of the output assembly (*.exe* or *.dll*) by using the `-out` compiler option. For more information, see **OutputAssembly** (C# compiler options) or -out (Visual Basic).

7. Run the *friend_signed_B.exe* file.

   The program outputs the string **Class1.Test**.

.NET security

There are similarities between the InternalsVisibleToAttribute attribute and the StrongNameIdentityPermission class. The main difference is that StrongNameIdentityPermission can demand security permissions to run a particular section of code, whereas the InternalsVisibleToAttribute attribute controls the visibility of `internal` (C#) or `Friend` (Visual Basic) types and members.

See also

- InternalsVisibleToAttribute
- Assemblies in .NET
- Friend assemblies
- How to: Create unsigned friend assemblies
- **KeyFile** (C#)
- -keyfile (Visual Basic)
- Sn.exe (Strong Name tool)
- Create and use strong-named assemblies
- C# programming guide
- Programming concepts (Visual Basic)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

How to: Determine if a file is an assembly

Article • 12/21/2022

A file is an assembly if and only if it is managed, and contains an assembly entry in its metadata. For more information on assemblies and metadata, see Assembly manifest.

How to manually determine if a file is an assembly

1. Start the Ildasm.exe (IL Disassembler).
2. Load the file you want to test.
3. If **ILDASM** reports that the file is not a portable executable (PE) file, then it is not an assembly. For more information, see the topic How to: View assembly contents.

How to programmatically determine if a file is an assembly

Using the AssemblyName class

1. Call the AssemblyName.GetAssemblyName method, passing the full file path and name of the file you are testing.
2. If a BadImageFormatException exception is thrown, the file is not an assembly.

This example tests a DLL to see if it is an assembly.

C#

```
using System;
using System.IO;
using System.Reflection;
using System.Runtime.InteropServices;

static class ExampleAssemblyName
{
    public static void CheckAssembly()
    {
        try
        {
```

```
            string path = Path.Combine(
                RuntimeEnvironment.GetRuntimeDirectory(),
                "System.Net.dll");

            AssemblyName testAssembly = AssemblyName.GetAssemblyName(path);
            Console.WriteLine("Yes, the file is an assembly.");
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("The file cannot be found.");
        }
        catch (BadImageFormatException)
        {
            Console.WriteLine("The file is not an assembly.");
        }
        catch (FileLoadException)
        {
            Console.WriteLine("The assembly has already been loaded.");
        }
    }

    /* Output:
    Yes, the file is an assembly.
    */
}
```

The GetAssemblyName method loads the test file, and then releases it once the information is read.

Using the PEReader class

1. If you're targeting .NET Standard or .NET Framework, install the System.Reflection.Metadata NuGet package. (When targeting .NET Core or .NET 5+, this step isn't required because this library is included in the shared framework.)

2. Create a System.IO.FileStream instance to read data from the file you're testing.

3. Create a System.Reflection.PortableExecutable.PEReader instance, passing your file stream into the constructor.

4. Check the value of the HasMetadata property. If the value is `false`, the file is not an assembly.

5. Call the GetMetadataReader method on the PE reader instance to create a metadata reader.

6. Check the value of the IsAssembly property. If the value is `true`, the file is an assembly.

Unlike the GetAssemblyName method, the PEReader class does not throw an exception on native Portable Executable (PE) files. This enables you to avoid the extra performance cost caused by exceptions when you need to check such files. You still need to handle exceptions in case the file does not exist or is not a PE file.

This example shows how to determine if a file is an assembly using the PEReader class.

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Reflection.Metadata;
using System.Reflection.PortableExecutable;
using System.Runtime.InteropServices;

static class ExamplePeReader
{
    static bool IsAssembly(string path)
    {
        using var fs = new FileStream(path, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite);

        // Try to read CLI metadata from the PE file.
        using var peReader = new PEReader(fs);

        if (!peReader.HasMetadata)
        {
            return false; // File does not have CLI metadata.
        }

        // Check that file has an assembly manifest.
        MetadataReader reader = peReader.GetMetadataReader();
        return reader.IsAssembly;
    }

    public static void CheckAssembly()
    {
        string path = Path.Combine(
                RuntimeEnvironment.GetRuntimeDirectory(),
                "System.Net.dll");

        try
        {
            if (IsAssembly(path))
            {
                Console.WriteLine("Yes, the file is an assembly.");
            }
            else
```

```
            {
                Console.WriteLine("The file is not an assembly.");
            }
        }
        catch (BadImageFormatException)
        {
            Console.WriteLine("The file is not an executable.");
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("The file cannot be found.");
        }
    }

    /* Output:
    Yes, the file is an assembly.
    */
}
```

See also

- AssemblyName
- C# programming guide
- Programming concepts (Visual Basic)
- Assemblies in .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to: Load and unload assemblies

Article • 09/15/2021

The assemblies referenced by your program will automatically be loaded by the common language runtime, but it is also possible to dynamically load specific assemblies into the current application domain. For more information, see How to: Load assemblies into an application domain.

In .NET Framework, there is no way to unload an individual assembly without unloading all of the application domains that contain it. Even if the assembly goes out of scope, the actual assembly file will remain loaded until all application domains that contain it are unloaded. In .NET Core, the System.Runtime.Loader.AssemblyLoadContext class handles the unloading of assemblies. For more information, see How to use and debug assembly unloadability in .NET Core.

Load and unload assemblies

To load an assembly into an application domain, use one of the several load methods contained in the classes AppDomain and Assembly. For more information, see How to: Load assemblies into an application domain. Note that .NET Core supports only a single application domain.

To unload an assembly in the .NET Framework, you must unload all of the application domains that contain it. To unload an application domain, use the AppDomain.Unload method. For more information, see How to: Unload an application domain.

If you want to unload some assemblies but not others in a .NET Framework application, consider creating a new application domain, executing the code inside that domain, and then unloading that application domain. For more information, see How to: Unload an application domain.

See also

- C# programming guide
- Programming concepts (Visual Basic)
- Assemblies in .NET
- How to: Load assemblies into an application domain

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Walkthrough: Embed types from managed assemblies in Visual Studio

Article • 10/14/2022

If you embed type information from a strong-named managed assembly, you can loosely couple types in an application to achieve version independence. That is, your program can be written to use types from any version of a managed library without having to be recompiled for each new version.

Type embedding is frequently used with COM interop, such as an application that uses automation objects from Microsoft Office. Embedding type information enables the same build of a program to work with different versions of Microsoft Office on different computers. However, you can also use type embedding with fully managed solutions.

After you specify the public interfaces that can be embedded, you create runtime classes that implement those interfaces. A client program can embed the type information for the interfaces at design time by referencing the assembly that contains the public interfaces and setting the `Embed Interop Types` property of the reference to `True`. The client program can then load instances of the runtime objects typed as those interfaces. This is equivalent to using the command line compiler and referencing the assembly by using the **EmbedInteropTypes** compiler option.

If you create a new version of your strong-named runtime assembly, the client program doesn't have to be recompiled. The client program continues to use whichever version of the runtime assembly is available to it, using the embedded type information for the public interfaces.

In this walkthrough, you:

1. Create a strong-named assembly with a public interface containing type information that can be embedded.
2. Create a strong-named runtime assembly that implements the public interface.
3. Create a client program that embeds the type information from the public interface and creates an instance of the class from the runtime assembly.
4. Modify and rebuild the runtime assembly.
5. Run the client program to see that it uses the new version of the runtime assembly without having to be recompiled.

> **Note**

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see **Personalizing the IDE**.

Conditions and limitations

You can embed type information from an assembly under the following conditions:

- The assembly exposes at least one public interface.
- The embedded interfaces are annotated with `ComImport` attributes and `Guid` attributes with unique GUIDs.
- The assembly is annotated with the `ImportedFromTypeLib` attribute or the `PrimaryInteropAssembly` attribute, and an assembly-level `Guid` attribute. The Visual C# and Visual Basic project templates include an assembly-level `Guid` attribute by default.

Because the primary function of type embedding is to support COM interop assemblies, the following limitations apply when you embed type information in a fully-managed solution:

- Only attributes specific to COM interop are embedded. Other attributes are ignored.
- If a type uses generic parameters, and the type of the generic parameter is an embedded type, that type cannot be used across an assembly boundary. Examples of crossing an assembly boundary include calling a method from another assembly or deriving a type from a type defined in another assembly.
- Constants are not embedded.
- The System.Collections.Generic.Dictionary<TKey,TValue> class does not support an embedded type as a key. You can implement your own dictionary type to support an embedded type as a key.

Create an interface

The first step is to create the type equivalence interface assembly.

1. In Visual Studio, select **File** > **New** > **Project**.

2. In the **Create a new project** dialog box, type *class library* in the **Search for templates** box. Select either the C# or Visual Basic **Class Library (.NET Framework)** template from the list, and then select **Next**.

3. In the **Configure your new project** dialog box, under **Project name**, type *TypeEquivalenceInterface*, and then select **Create**. The new project is created.

4. In **Solution Explorer**, right-click the *Class1.cs* or *Class1.vb* file, select **Rename**, and rename the file from *Class1* to *ISampleInterface*. Respond **Yes** to the prompt to also rename the class to `ISampleInterface`. This class represents the public interface for the class.

5. In **Solution Explorer**, right-click the **TypeEquivalenceInterface** project, and then select **Properties**.

6. Select **Build** on the left pane of the **Properties** screen, and set the **Output path** to a location on your computer, such as *C:\TypeEquivalenceSample*. You use the same location throughout this walkthrough.

7. Select **Build** > **Strong naming** on the left pane of the **Properties** screen, and then select the **Sign the assembly** check box. In the **Strong name key file**, select **Browse**.

8. Navigate to and select the *key.snk* file you created in the *TypeEquivalenceInterface* project, and then select **OK**. For more information, see Create a public-private key pair.

9. Open the *ISampleInterface* class file in the code editor, and replace its contents with the following code to create the `ISampleInterface` interface:

```
C#

using System;
using System.Runtime.InteropServices;

namespace TypeEquivalenceInterface
{
    [ComImport]
    [Guid("8DA56996-A151-4136-B474-32784559F6DF")]
    public interface ISampleInterface
    {
        void GetUserInput();
        string UserInput { get; }
    }
}
```

10. On the **Tools** menu, select **Create Guid**, and in the **Create GUID** dialog box, select **Registry Format**. Select **Copy**, and then select **Exit**.

11. In the `Guid` attribute of your code, replace the sample GUID with the GUID you copied, and remove the braces ({ }).

12. In **Solution Explorer**, expand the **Properties** folder and select the *AssemblyInfo.cs* or *AssemblyInfo.vb* file. In the code editor, add the following attribute to the file:

    C#

    ```
    [assembly: ImportedFromTypeLib("")]
    ```

13. Select **File** > **Save All** or press `Ctrl`+`Shift`+`S` to save the files and project.

14. In **Solution Explorer**, right-click the **TypeEquivalenceInterface** project and select **Build**. The class library DLL file is compiled and saved to the specified build output path, for example *C:\TypeEquivalenceSample*.

Create a runtime class

Next, create the type equivalence runtime class.

1. In Visual Studio, select **File** > **New** > **Project**.

2. In the **Create a new project** dialog box, type *class library* in the **Search for templates** box. Select either the C# or Visual Basic **Class Library (.NET Framework)** template from the list, and then select **Next**.

3. In the **Configure your new project** dialog box, under **Project name**, type *TypeEquivalenceRuntime*, and then select **Create**. The new project is created.

4. In **Solution Explorer**, right-click the *Class1.cs* or *Class1.vb* file, select **Rename**, and rename the file from *Class1* to *SampleClass*. Respond **Yes** to the prompt to also rename the class to `SampleClass`. This class implements the `ISampleInterface` interface.

5. In **Solution Explorer**, right-click the **TypeEquivalenceInterface** project and select **Properties**.

6. Select **Build** on the left pane of the **Properties** screen, and then set the **Output path** to the same location you used for the *TypeEquivalenceInterface* project, for example, *C:\TypeEquivalenceSample*.

7. Select **Build** > **Strong naming** on the left pane of the **Properties** screen, and then select the **Sign the assembly** check box. In the **Strong name key file**, select **Browse**.

8. Navigate to and select the *key.snk* file you created in the *TypeEquivalenceInterface* project, and then select **OK**. For more information, see Create a public-private key pair.

9. In **Solution Explorer**, right-click the **TypeEquivalenceRuntime** project and select **Add** > **Reference**.

10. In the **Reference Manager** dialog, select **Browse** and browse to the output path folder. Select the *TypeEquivalenceInterface.dll* file, select **Add**, and then select **OK**.

11. In **Solution Explorer**, expand the **References** folder and select the **TypeEquivalenceInterface** reference. In the **Properties** pane, set **Specific Version** to **False** if it is not already.

12. Open the *SampleClass* class file in the code editor, and replace its contents with the following code to create the `SampleClass` class:

    C#

    ```
    using System;
    using TypeEquivalenceInterface;

    namespace TypeEquivalenceRuntime
    {
        public class SampleClass : ISampleInterface
        {
            private string p_UserInput;
            public string UserInput { get { return p_UserInput; } }

            public void GetUserInput()
            {
                Console.WriteLine("Please enter a value:");
                p_UserInput = Console.ReadLine();
            }
        }
    }
    ```

13. Select **File** > **Save All** or press `Ctrl`+`Shift`+`S` to save the files and project.

14. In **Solution Explorer**, right-click the **TypeEquivalenceRuntime** project and select **Build**. The class library DLL file is compiled and saved to the specified build output path.

Create a client project

Finally, create a type equivalence client program that references the interface assembly.

1. In Visual Studio, select **File** > **New** > **Project**.

2. In the **Create a new project** dialog box, type *console* in the **Search for templates** box. Select either the C# or Visual Basic **Console App (.NET Framework)** template from the list, and then select **Next**.

3. In the **Configure your new project** dialog box, under **Project name**, type *TypeEquivalenceClient*, and then select **Create**. The new project is created.

4. In **Solution Explorer**, right-click the **TypeEquivalenceClient** project and select **Properties**.

5. Select **Build** on the left pane of the **Properties** screen, and then set the **Output path** to the same location you used for the TypeEquivalenceInterface project, for example, *C:\TypeEquivalenceSample*.

6. In **Solution Explorer**, right-click the **TypeEquivalenceClient** project and select **Add** > **Reference**.

7. In the **Reference Manager** dialog, if the **TypeEquivalenceInterface.dll** file is already listed, select it. If not, select **Browse**, browse to the output path folder, select the *TypeEquivalenceInterface.dll* file (not the *TypeEquivalenceRuntime.dll*), and select **Add**. Select **OK**.

8. In **Solution Explorer**, expand the **References** folder and select the **TypeEquivalenceInterface** reference. In the **Properties** pane, set **Embed Interop Types** to **True**.

9. Open the *Program.cs* or *Module1.vb* file in the code editor, and replace its contents with the following code to create the client program:

C#

```
using System;
using System.Reflection;
using TypeEquivalenceInterface;

namespace TypeEquivalenceClient
{
    class Program
    {
        static void Main(string[] args)
        {
            Assembly sampleAssembly =
Assembly.Load("TypeEquivalenceRuntime");
            ISampleInterface sampleClass =

(ISampleInterface)sampleAssembly.CreateInstance("TypeEquivalenceRuntime
```

```
.SampleClass");
            sampleClass.GetUserInput();
            Console.WriteLine(sampleClass.UserInput);

Console.WriteLine(sampleAssembly.GetName().Version.ToString());
            Console.ReadLine();
        }
    }
}
```

10. Select **File** > **Save All** or press `Ctrl`+`Shift`+`S` to save the files and project.

11. Press `Ctrl`+`F5` to build and run the program. Note that the console output returns the assembly version **1.0.0.0**.

Modify the interface

Now, modify the interface assembly, and change its version.

1. In Visual Studio, select **File** > **Open** > **Project/Solution**, and open the **TypeEquivalenceInterface** project.

2. In **Solution Explorer**, right-click the **TypeEquivalenceInterface** project and select **Properties**.

3. Select **Application** on the left pane of the **Properties** screen, and then select **Assembly Information**.

4. In the **Assembly Information** dialog box, change the **Assembly version** and **File version** values to *2.0.0.0*, and then select **OK**.

5. Open the *SampleInterface.cs* or *SampleInterface.vb* file, and add the following line of code to the `ISampleInterface` interface:

```
C#

DateTime GetDate();
```

6. Select **File** > **Save All** or press `Ctrl`+`Shift`+`S` to save the files and project.

7. In **Solution Explorer**, right-click the **TypeEquivalenceInterface** project and select **Build**. A new version of the class library DLL file is compiled and saved to the build output path.

Modify the runtime class

Also modify the runtime class and update its version.

1. In Visual Studio, select **File** > **Open** > **Project/Solution**, and open the **TypeEquivalenceRuntime** project.
2. In **Solution Explorer**, right-click the **TypeEquivalenceRuntime** project and select **Properties**.
3. Select **Application** on the left pane of the **Properties** screen, and then select **Assembly Information**.
4. In the **Assembly Information** dialog box, change the **Assembly version** and **File version** values to *2.0.0.0*, and then select **OK**.
5. Open the *SampleClass.cs* or *SampleClass.vb* file, and add the following code to the `SampleClass` class:

   C#

   ```
   public DateTime GetDate()
   {
       return DateTime.Now;
   }
   ```

6. Select **File** > **Save All** or press `Ctrl`+`Shift`+`S` to save the files and project.
7. In **Solution Explorer**, right-click the **TypeEquivalenceRuntime** project and select **Build**. A new version of the class library DLL file is compiled and saved to the build output path.

Run the updated client program

Go to the build output folder location and run *TypeEquivalenceClient.exe*. Note that the console output now reflects the new version of the `TypeEquivalenceRuntime` assembly, *2.0.0.0*, without the program being recompiled.

See also

- **EmbedInteropTypes** (C# Compiler Options)
- -link (Visual Basic)
- C# programming guide

- Programming concepts (Visual Basic)
- Assemblies in .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to: Inspect assembly contents using MetadataLoadContext

Article • 06/30/2022

The reflection API in .NET by default enables developers to inspect the contents of assemblies loaded into the main execution context. However, sometimes it isn't possible to load an assembly into the execution context, for example, because it was compiled for another platform or processor architecture, or it's a reference assembly. The System.Reflection.MetadataLoadContext API allows you to load and inspect such assemblies. Assemblies loaded into the MetadataLoadContext are treated only as metadata, that is, you can examine types in the assembly, but you can't execute any code contained in it. Unlike the main execution context, the MetadataLoadContext doesn't automatically load dependencies from the current directory; instead it uses the custom binding logic provided by the MetadataAssemblyResolver passed to it.

Prerequisites

To use MetadataLoadContext, install the System.Reflection.MetadataLoadContext NuGet package. It is supported on any .NET Standard 2.0-compliant target framework, for example, .NET Core 2.0 or .NET Framework 4.6.1.

Create MetadataAssemblyResolver for MetadataLoadContext

Creating the MetadataLoadContext requires providing the instance of the MetadataAssemblyResolver. The simplest way to provide one is to use the PathAssemblyResolver, which resolves assemblies from the given collection of assembly path strings. This collection, besides assemblies you want to inspect directly, should also include all needed dependencies. For example, to read the custom attribute located in an external assembly, you should include that assembly or an exception will be thrown. In most cases, you should include at least the *core assembly*, that is, the assembly containing built-in system types, such as System.Object. The following code shows how to create the PathAssemblyResolver using the collection consisting of the inspected assembly and the current runtime's core assembly:

C#

```
var resolver = new PathAssemblyResolver(new string[] {
```

```
"ExampleAssembly.dll", typeof(object).Assembly.Location });
```

If you need access to all BCL types, you can include all runtime assemblies in the collection. The following code shows how to create the PathAssemblyResolver using the collection consisting of the inspected assembly and all assemblies of the current runtime:

C#

```
// Get the array of runtime assemblies.
string[] runtimeAssemblies =
Directory.GetFiles(RuntimeEnvironment.GetRuntimeDirectory(), "*.dll");

// Create the list of assembly paths consisting of runtime assemblies and
the inspected assembly.
var paths = new List<string>(runtimeAssemblies);
paths.Add("ExampleAssembly.dll");

// Create PathAssemblyResolver that can resolve assemblies using the created
list.
var resolver = new PathAssemblyResolver(paths);
```

Create MetadataLoadContext

To create the MetadataLoadContext, invoke its constructor MetadataLoadContext(MetadataAssemblyResolver, String), passing the previously created MetadataAssemblyResolver as the first parameter and the core assembly name as the second parameter. You can omit the core assembly name, in which case the constructor will attempt to use default names: "mscorlib", "System.Runtime", or "netstandard".

After you've created the context, you can load assemblies into it using methods such as LoadFromAssemblyPath. You can use all reflection APIs on loaded assemblies except ones that involve code execution. The GetCustomAttributes method does involve the execution of constructors, so use the GetCustomAttributesData method instead when you need to examine custom attributes in the MetadataLoadContext.

The following code sample creates MetadataLoadContext, loads the assembly into it, and outputs assembly attributes into the console:

C#

```
var mlc = new MetadataLoadContext(resolver);

using (mlc)
```

```
{
    // Load assembly into MetadataLoadContext.
    Assembly assembly = mlc.LoadFromAssemblyPath("ExampleAssembly.dll");
    AssemblyName name = assembly.GetName();

    // Print assembly attribute information.
    Console.WriteLine($"{name.Name} has following attributes: ");

    foreach (CustomAttributeData attr in assembly.GetCustomAttributesData())
    {
        try
        {
            Console.WriteLine(attr.AttributeType);
        }
        catch (FileNotFoundException ex)
        {
            // We are missing the required dependency assembly.
            Console.WriteLine($"Error while getting attribute type:
{ex.Message}");
        }
    }
}
```

If you need to test types in MetadataLoadContext for equality or assignability, only use type objects loaded into that context. Mixing MetadataLoadContext types with runtime types is not supported. For example, consider a type `testedType` in MetadataLoadContext. If you need to test whether another type is assignable from it, don't use code like `typeof(MyType).IsAssignableFrom(testedType)`. Use code like this instead:

C#

```
Assembly matchAssembly =
mlc.LoadFromAssemblyPath(typeof(MyType).Assembly.Location);
Type matchType = assembly.GetType(typeof(MyType).FullName!)!;

if (matchType.IsAssignableFrom(testedType))
{
    Console.WriteLine($"{nameof(matchType)} is assignable from
{nameof(testedType)}");
}
```

Example

For a complete code example, see the Inspect assembly contents using MetadataLoadContext sample.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Metadata and Self-Describing Components

Article • 03/11/2022

In the past, a software component (.exe or .dll) that was written in one language could not easily use a software component that was written in another language. COM provided a step towards solving this problem. .NET makes component interoperation even easier by allowing compilers to emit additional declarative information into all modules and assemblies. This information, called metadata, helps components to interact seamlessly.

Metadata is binary information describing your program that is stored either in a common language runtime portable executable (PE) file or in memory. When you compile your code into a PE file, metadata is inserted into one portion of the file, and your code is converted to Microsoft intermediate language (MSIL) and inserted into another portion of the file. Every type and member that is defined and referenced in a module or assembly is described within metadata. When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on.

Metadata describes every type and member defined in your code in a language-neutral manner. Metadata stores the following information:

- Description of the assembly.
    - Identity (name, version, culture, public key).
    - The types that are exported.
    - Other assemblies that this assembly depends on.
    - Security permissions needed to run.
- Description of types.
    - Name, visibility, base class, and interfaces implemented.
    - Members (methods, fields, properties, events, nested types).
- Attributes.
    - Additional descriptive elements that modify types and members.

Benefits of Metadata

Metadata is the key to a simpler programming model, and eliminates the need for Interface Definition Language (IDL) files, header files, or any external method of component reference. Metadata enables .NET languages to describe themselves automatically in a language-neutral manner, unseen by both the developer and the user. Additionally, metadata is extensible through the use of attributes. Metadata provides the following major benefits:

- Self-describing files.

  Common language runtime modules and assemblies are self-describing. A module's metadata contains everything needed to interact with another module. Metadata automatically provides the functionality of IDL in COM, so you can use one file for both definition and implementation. Runtime modules and assemblies do not even require registration with the operating system. As a result, the descriptions used by the runtime always reflect the actual code in your compiled file, which increases application reliability.

- Language interoperability and easier component-based design.

  Metadata provides all the information required about compiled code for you to inherit a class from a PE file written in a different language. You can create an instance of any class written in any managed language (any language that targets the common language runtime) without worrying about explicit marshalling or using custom interoperability code.

- Attributes.

  .NET lets you declare specific kinds of metadata, called attributes, in your compiled file. Attributes can be found throughout .NET and are used to control in more detail how your program behaves at run time. Additionally, you can emit your own custom metadata into .NET files through user-defined custom attributes. For more information, see Attributes.

Metadata and the PE File Structure

Metadata is stored in one section of a .NET portable executable (PE) file, while Microsoft intermediate language (MSIL) is stored in another section of the PE file. The metadata portion of the file contains a series of table and heap data structures. The MSIL portion contains MSIL and metadata tokens that reference the metadata portion of the PE file.

You might encounter metadata tokens when you use tools such as the MSIL Disassembler (Ildasm.exe) to view your code's MSIL, for example.

Metadata Tables and Heaps

Each metadata table holds information about the elements of your program. For example, one metadata table describes the classes in your code, another table describes the fields, and so on. If you have ten classes in your code, the class table will have tens rows, one for each class. Metadata tables reference other tables and heaps. For example, the metadata table for classes references the table for methods.

Metadata also stores information in four heap structures: string, blob, user string, and GUID. All the strings used to name types and members are stored in the string heap. For example, a method table does not directly store the name of a particular method, but points to the method's name stored in the string heap.

Metadata Tokens

Each row of each metadata table is uniquely identified in the MSIL portion of the PE file by a metadata token. Metadata tokens are conceptually similar to pointers, persisted in MSIL, that reference a particular metadata table.

A metadata token is a four-byte number. The top byte denotes the metadata table to which a particular token refers (method, type, and so on). The remaining three bytes specify the row in the metadata table that corresponds to the programming element being described. If you define a method in C# and compile it into a PE file, the following metadata token might exist in the MSIL portion of the PE file:

`0x06000004`

The top byte (`0x06`) indicates that this is a **MethodDef** token. The lower three bytes (`000004`) tells the common language runtime to look in the fourth row of the **MethodDef** table for the information that describes this method definition.

Metadata within a PE File

When a program is compiled for the common language runtime, it is converted to a PE file that consists of three parts. The following table describes the contents of each part.

Expand table

PE section	Contents of PE section
PE header	The index of the PE file's main sections and the address of the entry point. The runtime uses this information to identify the file as a PE file and to determine where execution starts when loading the program into memory.
MSIL instructions	The Microsoft intermediate language instructions (MSIL) that make up your code. Many MSIL instructions are accompanied by metadata tokens.
Metadata	Metadata tables and heaps. The runtime uses this section to record information about every type and member in your code. This section also includes custom attributes and security information.

Run-Time Use of Metadata

To better understand metadata and its role in the common language runtime, it might be helpful to construct a simple program and illustrate how metadata affects its run-time life. The following code example shows two methods inside a class called `MyApp`. The `Main` method is the program entry point, while the `Add` method simply returns the sum of two integer arguments.

C#

```
using System;
public class MyApp
{
   public static int Main()
   {
      int ValueOne = 10;
      int ValueTwo = 20;
      Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo));
      return 0;
   }
   public static int Add(int One, int Two)
   {
      return (One + Two);
   }
}
```

When the code runs, the runtime loads the module into memory and consults the metadata for this class. Once loaded, the runtime performs extensive analysis of the method's Microsoft intermediate language (MSIL) stream to convert it to fast native machine instructions. The runtime uses a just-in-time (JIT) compiler to convert the MSIL instructions to native machine code one method at a time as needed.

The following example shows part of the MSIL produced from the previous code's `Main` function. You can view the MSIL and metadata from any .NET application using the MSIL Disassembler (Ildasm.exe).

Console

```
.entrypoint
.maxstack  3
.locals ([0] int32 ValueOne,
         [1] int32 ValueTwo,
         [2] int32 V_2,
         [3] int32 V_3)
IL_0000:  ldc.i4.s   10
IL_0002:  stloc.0
IL_0003:  ldc.i4.s   20
IL_0005:  stloc.1
IL_0006:  ldstr      "The Value is: {0}"
IL_000b:  ldloc.0
IL_000c:  ldloc.1
IL_000d:  call int32 ConsoleApplication.MyApp::Add(int32,int32) /* 06000003
*/
```

The JIT compiler reads the MSIL for the whole method, analyzes it thoroughly, and generates efficient native instructions for the method. At `IL_000d`, a metadata token for the `Add` method (`/* 06000003 */`) is encountered and the runtime uses the token to consult the third row of the **MethodDef** table.

The following table shows part of the **MethodDef** table referenced by the metadata token that describes the `Add` method. While other metadata tables exist in this assembly and have their own unique values, only this table is discussed.

Expand table

Row	Relative Virtual Address (RVA)	ImplFlags	Flags	Name (Points to string heap.)	Signature (Points to blob heap.)
1	0x00002050	IL Managed	Public ReuseSlot SpecialName RTSpecialName .ctor	.ctor (constructor)	

Row	Relative Virtual Address (RVA)	ImplFlags	Flags	Name (Points to string heap.)	Signature (Points to blob heap.)
2	0x00002058	IL Managed	Public Static ReuseSlot	Main	String
3	0x0000208c	IL Managed	Public Static ReuseSlot	Add	int, int, int

Each column of the table contains important information about your code. The **RVA** column allows the runtime to calculate the starting memory address of the MSIL that defines this method. The **ImplFlags** and **Flags** columns contain bitmasks that describe the method (for example, whether the method is public or private). The **Name** column indexes the name of the method from the string heap. The **Signature** column indexes the definition of the method's signature in the blob heap.

The runtime calculates the desired offset address from the **RVA** column in the third row and returns this address to the JIT compiler, which then proceeds to the new address. The JIT compiler continues to process MSIL at the new address until it encounters another metadata token and the process is repeated.

Using metadata, the runtime has access to all the information it needs to load your code and process it into native machine instructions. In this manner, metadata enables self-describing files and, together with the common type system, cross-language inheritance.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

System.Reflection.Emit.AssemblyBuilder class

Article • 02/14/2024

This article provides supplementary remarks to the reference documentation for this API.

A dynamic assembly is an assembly that is created using the Reflection Emit APIs. A dynamic assembly can reference types defined in another dynamic or static assembly. You can use AssemblyBuilder to generate dynamic assemblies in memory and execute their code during the same application run. .NET 9 added a fully managed implementation for reflection emit that allows you save the assembly into a file. In .NET Framework, you can do both—run the dynamic assembly and save it to a file. The dynamic assembly created for saving is called a *persistable* assembly, while the regular memory-only assembly is called *transient*. In .NET Framework, a dynamic assembly can consist of one or more dynamic modules. In .NET Core and .NET 5+, a dynamic assembly can only consist of one dynamic module.

The way you create an AssemblyBuilder instance differs for each implementation, but further steps for defining a module, type, method, or enum, and for writing IL, are quite similar.

Runnable dynamic assemblies in .NET Core

To get a runnable AssemblyBuilder object, use the AssemblyBuilder.DefineDynamicAssembly method. Dynamic assemblies can be created using one of the following access modes:

- AssemblyBuilderAccess.Run

  The dynamic assembly represented by an AssemblyBuilder can be used to execute the emitted code.

- AssemblyBuilderAccess.RunAndCollect

  The dynamic assembly represented by an AssemblyBuilder can be used to execute the emitted code and is automatically reclaimed by garbage collector.

The access mode must be specified by providing the appropriate AssemblyBuilderAccess value in the call to the AssemblyBuilder.DefineDynamicAssembly method when the dynamic assembly is defined and cannot be changed later. The

runtime uses the access mode of a dynamic assembly to optimize the assembly's internal representation.

The following example demonstrates how to create and run an assembly:

C#

```
public void CreateAndRunAssembly(string assemblyPath)
{
    AssemblyBuilder ab = AssemblyBuilder.DefineDynamicAssembly(new
AssemblyName("MyAssembly"), AssemblyBuilderAccess.Run);
    ModuleBuilder mob = ab.DefineDynamicModule("MyModule");
    TypeBuilder tb = mob.DefineType("MyType", TypeAttributes.Public |
TypeAttributes.Class);
    MethodBuilder mb = tb.DefineMethod("SumMethod", MethodAttributes.Public
| MethodAttributes.Static,

typeof(int), new Type[] {typeof(int), typeof(int)});
    ILGenerator il = mb.GetILGenerator();
    il.Emit(OpCodes.Ldarg_0);
    il.Emit(OpCodes.Ldarg_1);
    il.Emit(OpCodes.Add);
    il.Emit(OpCodes.Ret);

    Type type = tb.CreateType();

    MethodInfo method = type.GetMethod("SumMethod");
    Console.WriteLine(method.Invoke(null, new object[] { 5, 10 }));
}
```

Persistable dynamic assemblies in .NET Core

The AssemblyBuilder.Save API wasn't originally ported to .NET (Core) because the implementation depended heavily on Windows-specific native code that also wasn't ported. However, because you could only *run* a generated assembly and not *save* it, it was difficult to debug these in-memory assemblies. Other advantages of saving a dynamic assembly to a file are:

- You can verify the generated assembly with tools such as ILVerify, or decompile and manually examine it with tools such as ILSpy.
- The saved assembly can be shared or loaded directly, which can decrease application startup time.

.NET 9 adds a fully managed `Reflection.Emit` implementation that supports saving. This implementation has no dependency on the pre-existing, runtime-specific `Reflection.Emit` implementation. That is, now there are two different implementations.

The assemblies generated with the new managed implementation can be saved. To run the assembly, first save it into a memory stream or a file, then load it back.

To create a persistable `AssemblyBuilder` instance, use the static factory method AssemblyBuilder.DefinePersistedAssembly(AssemblyName, Assembly, IEnumerable<CustomAttributeBuilder>). The `coreAssembly` parameter is used to resolve base runtime types and can be used for resolving reference assembly versioning.

- If `Reflection.Emit` is used to generate an assembly that targets a specific TFM, open the reference assemblies for the given TFM using `MetadataLoadContext` and use the value of the MetadataLoadContext.CoreAssembly property for `coreAssembly`. This value allows the generator to run on one .NET runtime version and target a different .NET runtime version.

- If `Reflection.Emit` is used to generate an assembly that's only going to be executed on the same runtime version as the runtime version that the compiler is running on (typically in-proc), the core assembly can be `typeof(object).Assembly`. The reference assemblies aren't necessary in this case.

The following example demonstrates how to create and save an assembly to a stream and run it:

C#

```
public void CreateSaveAndRunAssembly(string assemblyPath)
{
    AssemblyBuilder ab = AssemblyBuilder.DefinePersistedAssembly(new
AssemblyName("MyAssembly"), typeof(object).Assembly);
    ModuleBuilder mob = ab.DefineDynamicModule("MyModule");
    TypeBuilder tb = mob.DefineType("MyType", TypeAttributes.Public |
TypeAttributes.Class);
    MethodBuilder meb = tb.DefineMethod("SumMethod", MethodAttributes.Public
| MethodAttributes.Static,

typeof(int), new Type[] {typeof(int), typeof(int)});
    ILGenerator il = meb.GetILGenerator();
    il.Emit(OpCodes.Ldarg_0);
    il.Emit(OpCodes.Ldarg_1);
    il.Emit(OpCodes.Add);
    il.Emit(OpCodes.Ret);

    tb.CreateType();

    using var stream = new MemoryStream();
    ab.Save(stream);  // or pass filename to save into a file
    stream.Seek(0, SeekOrigin.Begin);
    Assembly assembly = AssemblyLoadContext.Default.LoadFromStream(stream);
    MethodInfo method = assembly.GetType("MyType").GetMethod("SumMethod");
```

```
    Console.WriteLine(method.Invoke(null, new object[] { 5, 10 }));
}
```

> **Note**
>
> The metadata tokens for all members are populated on the **Save** operation. Don't use the tokens of a generated type and its members before saving, as they'll have default values or throw exceptions. It's safe to use tokens for types that are referenced, not generated.
>
> Some APIs that aren't important for emitting an assembly aren't implemented; for example, `GetCustomAttributes()` is not implemented. With the runtime implementation, you were able to use those APIs after creating the type. For the persisted `AssemblyBuilder`, they throw `NotSupportedException` or `NotImplementedException`. If you have a scenario that requires those APIs, file an issue in the **dotnet/runtime repo**.

For an alternative way to generate assembly files, see MetadataBuilder.

Persistable dynamic assemblies in .NET Framework

In .NET Framework, dynamic assemblies and modules can be saved to files. To support this feature, the AssemblyBuilderAccess enumeration declares two additional fields: Save and RunAndSave.

The dynamic modules in the persistable dynamic assembly are saved when the dynamic assembly is saved using the Save method. To generate an executable, the SetEntryPoint method must be called to identify the method that is the entry point to the assembly. Assemblies are saved as DLLs by default, unless the SetEntryPoint method requests the generation of a console application or a Windows-based application.

The following example demonstrates how to create, save, and run an assembly using .NET Framework.

C#

```
public void CreateRunAndSaveAssembly(string assemblyPath)
{
    AssemblyBuilder ab = Thread.GetDomain().DefineDynamicAssembly(new
AssemblyName("MyAssembly"), AssemblyBuilderAccess.RunAndSave);
    ModuleBuilder mob = ab.DefineDynamicModule("MyAssembly.dll");
```

```
    TypeBuilder tb = mob.DefineType("MyType", TypeAttributes.Public |
TypeAttributes.Class);
    MethodBuilder meb = tb.DefineMethod("SumMethod", MethodAttributes.Public
| MethodAttributes.Static,

typeof(int), new Type[] {typeof(int), typeof(int)});
    ILGenerator il = meb.GetILGenerator();
    il.Emit(OpCodes.Ldarg_0);
    il.Emit(OpCodes.Ldarg_1);
    il.Emit(OpCodes.Add);
    il.Emit(OpCodes.Ret);

    Type type = tb.CreateType();

    MethodInfo method = type.GetMethod("SumMethod");
    Console.WriteLine(method.Invoke(null, new object[] { 5, 10 }));
    ab.Save("MyAssembly.dll");
}
```

Some methods on the base Assembly class, such as `GetModules` and `GetLoadedModules`, won't work correctly when called from AssemblyBuilder objects. You can load the defined dynamic assembly and call the methods on the loaded assembly. For example, to ensure that resource modules are included in the returned module list, call `GetModules` on the loaded Assembly object. If a dynamic assembly contains more than one dynamic module, the assembly's manifest file name should match the module's name that's specified as the first argument to the DefineDynamicModule method.

The signing of a dynamic assembly using KeyPair is not effective until the assembly is saved to disk. So, strong names will not work with transient dynamic assemblies.

Dynamic assemblies can reference types defined in another assembly. A transient dynamic assembly can safely reference types defined in another transient dynamic assembly, a persistable dynamic assembly, or a static assembly. However, the common language runtime does not allow a persistable dynamic module to reference a type defined in a transient dynamic module. This is because when the persisted dynamic module is loaded after being saved to disk, the runtime cannot resolve the references to types defined in the transient dynamic module.

Restrictions on emitting to remote application domains

Some scenarios require a dynamic assembly to be created and executed in a remote application domain. Reflection emit does not allow a dynamic assembly to be emitted directly to a remote application domain. The solution is to emit the dynamic assembly in the current application domain, save the emitted dynamic assembly to disk, and then

load the dynamic assembly into the remote application domain. The remoting and application domains are supported only in .NET Framework.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

System.Reflection.Emit.MethodBuilder class

Article • 01/09/2024

This article provides supplementary remarks to the reference documentation for this API.

The MethodBuilder class is used to fully describe a method in Microsoft intermediate language (MSIL), including the name, attributes, signature, and method body. It is used in conjunction with the TypeBuilder class to create classes at runtime.

You can use reflection emit to define global methods and to define methods as type members. The APIs that define methods return MethodBuilder objects.

Global methods

A global method is defined by using the ModuleBuilder.DefineGlobalMethod method, which returns a `MethodBuilder` object.

Global methods must be static. If a dynamic module contains global methods, the ModuleBuilder.CreateGlobalFunctions method must be called before persisting the dynamic module or the containing dynamic assembly because the common language runtime postpones fixing up the dynamic module until all global functions have been defined.

A global native method is defined by using the ModuleBuilder.DefinePInvokeMethod method. Platform invoke (PInvoke) methods must not be declared abstract or virtual. The runtime sets the MethodAttributes.PinvokeImpl attribute for a platform invoke method.

Methods as members of types

A method is defined as a type member by using the TypeBuilder.DefineMethod method, which returns a MethodBuilder object.

The DefineParameter method is used to set the name and parameter attributes of a parameter, or of the return value. The ParameterBuilder object returned by this method represents a parameter or the return value. The ParameterBuilder object can be used to set the marshaling, to set the constant value, and to apply custom attributes.

Attributes

Members of the MethodAttributes enumeration define the precise character of a dynamic method:

- Static methods are specified using the MethodAttributes.Static attribute.
- Final methods (methods that cannot be overridden) are specified using the MethodAttributes.Final attribute.
- Virtual methods are specified using the MethodAttributes.Virtual attribute.
- Abstract methods are specified using the MethodAttributes.Abstract attribute.
- Several attributes determine method visibility. See the description of the MethodAttributes enumeration.
- Methods that implement overloaded operators must set the MethodAttributes.SpecialName attribute.
- Finalizers must set the MethodAttributes.SpecialName attribute.

Known issues

- Although MethodBuilder is derived from MethodInfo, some of the abstract methods defined in the MethodInfo class are not fully implemented in MethodBuilder. These MethodBuilder methods throw the NotSupportedException. For example the MethodBuilder.Invoke method is not fully implemented. You can reflect on these methods by retrieving the enclosing type using the Type.GetType or Assembly.GetType methods.
- Custom modifiers are supported.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

System.Reflection.Emit.TypeBuilder class

Article • 01/09/2024

This article provides supplementary remarks to the reference documentation for this API.

TypeBuilder is the root class used to control the creation of dynamic classes in the runtime. It provides a set of routines that are used to define classes, add methods and fields, and create the class inside a module. A new TypeBuilder can be created from a dynamic module by calling the ModuleBuilder.DefineType method, which returns a TypeBuilder object.

Reflection emit provides the following options for defining types:

- Define a class or interface with the given name.
- Define a class or interface with the given name and attributes.
- Define a class with the given name, attributes, and base class.
- Define a class with the given name, attributes, base class, and the set of interfaces that the class implements.
- Define a class with the given name, attributes, base class, and packing size.
- Define a class with the given name, attributes, base class, and the class size as a whole.
- Define a class with the given name, attributes, base class, packing size, and the class size as a whole.

To create an array type, pointer type, or byref type for an incomplete type that is represented by a TypeBuilder object, use the MakeArrayType method, MakePointerType method, or MakeByRefType method, respectively.

Before a type is used, the TypeBuilder.CreateType method must be called. **CreateType** completes the creation of the type. Following the call to **CreateType**, the caller can instantiate the type by using the Activator.CreateInstance method, and invoke members of the type by using the Type.InvokeMember method. It is an error to invoke methods that change the implementation of a type after **CreateType** has been called. For example, the common language runtime throws an exception if the caller tries to add new members to a type.

A class initializer is created by using the TypeBuilder.DefineTypeInitializer method. **DefineTypeInitializer** returns a ConstructorBuilder object.

Nested types are defined by calling one of the TypeBuilder.DefineNestedType methods.

Attributes

The TypeBuilder class uses the TypeAttributes enumeration to further specify the characteristics of the type to be created:

- Interfaces are specified using the TypeAttributes.Interface and TypeAttributes.Abstract attributes.
- Concrete classes (classes that cannot be extended) are specified using the TypeAttributes.Sealed attribute.
- Several attributes determine type visibility. See the description of the TypeAttributes enumeration.
- If TypeAttributes.SequentialLayout is specified, the class loader lays out fields in the order they are read from metadata. The class loader considers the specified packing size but ignores any specified field offsets. The metadata preserves the order in which the field definitions are emitted. Even across a merge, the metadata will not reorder the field definitions. The loader will honor the specified field offsets only if TypeAttributes.ExplicitLayout is specified.

Known issues

- Reflection emit does not verify whether a non-abstract class that implements an interface has implemented all the methods declared in the interface. However, if the class does not implement all the methods declared in an interface, the runtime does not load the class.
- Although TypeBuilder is derived from Type, some of the abstract methods defined in the Type class are not fully implemented in the TypeBuilder class. Calls to these TypeBuilder methods throw a NotSupportedException exception. The desired functionality can be obtained by retrieving the created type using the Type.GetType or Assembly.GetType and reflecting on the retrieved type.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

System.Reflection.Emit.DynamicMethod class

Article • 01/09/2024

This article provides supplementary remarks to the reference documentation for this API.

You can use the DynamicMethod class to generate and execute a method at run time, without having to generate a dynamic assembly and a dynamic type to contain the method. The executable code created by the just-in-time (JIT) compiler is reclaimed when the DynamicMethod object is reclaimed. Dynamic methods are the most efficient way to generate and execute small amounts of code.

A dynamic method can be anonymously hosted, or it can be logically associated with a module or with a type.

- If the dynamic method is anonymously hosted, it is located in a system-provided assembly, and therefore is isolated from other code. By default, it does not have access to any non-public data. An anonymously hosted dynamic method can have restricted ability to skip the JIT compiler's visibility checks, if it has been granted ReflectionPermission with the ReflectionPermissionFlag.RestrictedMemberAccess flag. The trust level of the assembly whose non-public members are accessed by the dynamic method must be equal to, or a subset of, the trust level of the call stack that emitted the dynamic method. For more information about anonymously hosted dynamic methods, see Walkthrough: Emitting Code in Partial Trust Scenarios.

- If the dynamic method is associated with a module that you specify, the dynamic method is effectively global to that module. It can access all types in the module and all `internal` (`Friend` in Visual Basic) members of the types. You can associate a dynamic method with any module, regardless of whether you created the module, provided that a demand for ReflectionPermission with the RestrictedMemberAccess flag can be satisfied by the call stack that includes your code. If the ReflectionPermissionFlag.MemberAccess flag is included in the grant, the dynamic method can skip the JIT compiler's visibility checks and access the private data of all types declared in the module or in any other module in any assembly.

  > **Note**

When you specify the module with which a dynamic method is associated, that module must not be in the system-provided assembly that is used for anonymous hosting.

- If the dynamic method is associated with a type that you specify, it has access to all members of the type, regardless of access level. In addition, JIT visibility checks can be skipped. This gives the dynamic method access to the private data of other types declared in the same module or in any other module in any assembly. You can associate a dynamic method with any type, but your code must be granted ReflectionPermission with both the RestrictedMemberAccess and MemberAccess flags.

The following table shows which types and members are accessible to an anonymously hosted dynamic method, with and without JIT visibility checks, depending on whether ReflectionPermission with the RestrictedMemberAccess flag is granted.

Expand table

Visibility checks	**Without `RestrictedMemberAccess`**	**With `RestrictedMemberAccess`**
Without skipping JIT visibility checks	Public members of public types in any assembly.	Public members of public types in any assembly.
Skipping JIT visibility checks, with restrictions	Public members of public types in any assembly.	All members of all types, only in assemblies whose trust levels are equal to or less than the trust level of the assembly that emitted the dynamic method.

The following table shows which types and members are accessible to a dynamic method that's associated with a module or with a type in a module.

Expand table

Skip JIT visibility checks	**Associated with module**	**Associated with type**
No	Public and internal members of public, internal, and private types in the module. Public members of public types in any assembly.	All members of the associated type. Public and internal members of all the other types in the module. Public members of public types in any assembly.

Skip JIT visibility checks	Associated with module	Associated with type
Yes	All members of all types in any assembly.	All members of all types in any assembly.

A dynamic method that is associated with a module has the permissions of that module. A dynamic method that is associated with a type has the permissions of the module containing that type.

Dynamic methods and their parameters do not have to be named, but you can specify names to assist in debugging. Custom attributes are not supported on dynamic methods or their parameters.

Although dynamic methods are `static` methods (`Shared` methods in Visual Basic), the relaxed rules for delegate binding allow a dynamic method to be bound to an object, so that it acts like an instance method when called using that delegate instance. An example that demonstrates this is provided for the CreateDelegate(Type, Object) method overload.

Verification

The following list summarizes the conditions under which dynamic methods can contain unverifiable code. (For example, a dynamic method is unverifiable if its InitLocals property is set to `false`.)

- A dynamic method that's associated with a security-critical assembly is also security-critical, and can skip verification. For example, an assembly without security attributes that is run as a desktop application is treated as security-critical by the runtime. If you associate a dynamic method with the assembly, the dynamic method can contain unverifiable code.
- If a dynamic method that contains unverifiable code is associated with an assembly that has level 1 transparency, the just-in-time (JIT) compiler injects a security demand. The demand succeeds only if the dynamic method is executed by fully trusted code. See Security-Transparent Code, Level 1.
- If a dynamic method that contains unverifiable code is associated with an assembly that has level 2 transparency (such as mscorlib.dll), it throws an exception (injected by the JIT compiler) instead of making a security demand. See Security-Transparent Code, Level 2.
- An anonymously hosted dynamic method that contains unverifiable code always throws an exception. It can never skip verification, even if it is created and

executed by fully trusted code.

The exception that's thrown for unverifiable code varies depending on the way the dynamic method is invoked. If you invoke a dynamic method by using a delegate returned from the CreateDelegate method, a VerificationException is thrown. If you invoke the dynamic method by using the Invoke method, a TargetInvocationException is thrown with an inner VerificationException.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

System.Reflection.PortableExecutable.DebugDirectoryEntryType enum

Article • 01/09/2024

This article provides supplementary remarks to the reference documentation for this API.

The DebugDirectoryEntryType enum describes the format of the debugging information of a DebugDirectoryEntry.

See the following for the specifications related to individual enumeration members:

Expand table

Member	Specification
`CodeView`	CodeView Debug Directory Entry (type 2)
`EmbeddedPortablePdb`	Embedded Portable PDB Debug Directory Entry (type 17)
`PdbChecksum`	PDB Checksum Debug Directory Entry (type 19)
`Reproducible`	See Deterministic Debug Directory Entry (type 16)

DebugDirectoryEntryType.Reproducible

The tool that produced the deterministic PE/COFF file guarantees that the entire content of the file is based solely on documented inputs given to the tool (such as source files, resource files, compiler options, etc.) rather than ambient environment variables (such as the current time, the operating system, the bitness of the process running the tool, etc.).

The value of the `TimeDateStamp` field in the COFF File Header of a deterministic PE/COFF file does not indicate the date and time when the file was produced and should not be interpreted that way. Instead, the value of the field is derived from a hash of the file content. The algorithm to calculate this value is an implementation detail of the tool that produced the file.

The debug directory entry of type Reproducible must have all fields except for DebugDirectoryEntry.Type zeroed.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Dependency loading in .NET

Article • 08/30/2022

Every .NET application has dependencies. Even the simple `hello world` app has dependencies on portions of the .NET class libraries.

Understanding the default assembly loading logic in .NET can help you troubleshoot typical deployment issues.

In some applications, dependencies are dynamically determined at run time. In these situations, it's critical to understand how managed assemblies and unmanaged dependencies are loaded.

AssemblyLoadContext

The AssemblyLoadContext API is central to the .NET loading design. The Understanding AssemblyLoadContext article provides a conceptual overview of the design.

Loading details

The loading algorithm details are covered briefly in several articles:

- Managed assembly loading algorithm
- Satellite assembly loading algorithm
- Unmanaged (native) library loading algorithm
- Default probing

Create an app with plugins

The tutorial Create a .NET application with plugins describes how to create a custom AssemblyLoadContext. It uses an AssemblyDependencyResolver to resolve the dependencies of the plugin. The tutorial correctly isolates the plugin's dependencies from the hosting application.

Assembly unloadability

The How to use and debug assembly unloadability in .NET article is a step-by-step tutorial. It shows how to load a .NET application, execute it, and then unload it. The article also provides debugging tips.

Collect detailed assembly loading information

The Collect detailed assembly loading information article describes how to collect detailed information about managed assembly loading in the runtime. It uses the dotnet-trace tool to capture assembly loader events in a trace of a running process.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

About System.Runtime.Loader.AssemblyLoadContext

Article • 08/30/2022

The AssemblyLoadContext class was introduced in .NET Core and is not available in .NET Framework. This article supplements the AssemblyLoadContext API documentation with conceptual information.

This article is relevant to developers implementing dynamic loading, especially dynamic-loading framework developers.

What is the AssemblyLoadContext?

Every .NET 5+ and .NET Core application implicitly uses AssemblyLoadContext. It's the runtime's provider for locating and loading dependencies. Whenever a dependency is loaded, an AssemblyLoadContext instance is invoked to locate it.

- AssemblyLoadContext provides a service of locating, loading, and caching managed assemblies and other dependencies.
- To support dynamic code loading and unloading, it creates an isolated context for loading code and its dependencies in their own AssemblyLoadContext instance.

Versioning rules

A single AssemblyLoadContext instance is limited to loading exactly one version of an Assembly per simple assembly name. When an assembly reference is resolved against an AssemblyLoadContext instance that already has an assembly of that name loaded, the requested version is compared to the loaded version. The resolution will succeed only if the loaded version is equal or higher to the requested version.

When do you need multiple AssemblyLoadContext instances?

The restriction that a single AssemblyLoadContext instance can load only one version of an assembly can become a problem when loading code modules dynamically. Each module is independently compiled, and the modules may depend on different versions

of an Assembly. This is often a problem when different modules depend on different versions of a commonly used library.

To support dynamically loading code, the AssemblyLoadContext API provides for loading conflicting versions of an Assembly in the same application. Each AssemblyLoadContext instance provides a unique dictionary that maps each AssemblyName.Name to a specific Assembly instance.

It also provides a convenient mechanism for grouping dependencies related to a code module for later unload.

The AssemblyLoadContext.Default instance

The AssemblyLoadContext.Default instance is automatically populated by the runtime at startup. It uses default probing to locate and find all static dependencies.

It solves the most common dependency loading scenarios.

Dynamic dependencies

AssemblyLoadContext has various events and virtual functions that can be overridden.

The AssemblyLoadContext.Default instance only supports overriding the events.

The articles Managed assembly loading algorithm, Satellite assembly loading algorithm, and Unmanaged (native) library loading algorithm refer to all the available events and virtual functions. The articles show each event and function's relative position in the loading algorithms. This article doesn't reproduce that information.

This section covers the general principles for the relevant events and functions.

- **Be repeatable**. A query for a specific dependency must always result in the same response. The same loaded dependency instance must be returned. This requirement is fundamental for cache consistency. For managed assemblies in particular, we're creating an Assembly cache. The cache key is a simple assembly name, AssemblyName.Name.
- **Typically don't throw**. It's expected that these functions return `null` rather than throw when unable to find the requested dependency. Throwing will prematurely end the search and propagate an exception to the caller. Throwing should be restricted to unexpected errors like a corrupted assembly or an out of memory condition.

- **Avoid recursion**. Be aware that these functions and handlers implement the loading rules for locating dependencies. Your implementation shouldn't call APIs that trigger recursion. Your code should typically call **AssemblyLoadContext** load functions that require a specific path or memory reference argument.
- **Load into the correct AssemblyLoadContext**. The choice of where to load dependencies is application-specific. The choice is implemented by these events and functions. When your code calls **AssemblyLoadContext** load-by-path functions call them on the instance where you want the code loaded. Sometime returning `null` and letting the AssemblyLoadContext.Default handle the load may be the simplest option.
- **Be aware of thread races**. Loading can be triggered by multiple threads. The AssemblyLoadContext handles thread races by atomically adding assemblies to its cache. The race loser's instance is discarded. In your implementation logic, don't add extra logic that doesn't handle multiple threads properly.

How are dynamic dependencies isolated?

Each AssemblyLoadContext instance represents a unique scope for Assembly instances and Type definitions.

There's no binary isolation between these dependencies. They're only isolated by not finding each other by name.

In each AssemblyLoadContext:

- AssemblyName.Name may refer to a different Assembly instance.
- Type.GetType may return a different type instance for the same type `name`.

Shared dependencies

Dependencies can easily be shared between AssemblyLoadContext instances. The general model is for one AssemblyLoadContext to load a dependency. The other shares the dependency by using a reference to the loaded assembly.

This sharing is required of the runtime assemblies. These assemblies can only be loaded into the AssemblyLoadContext.Default. The same is required for frameworks like `ASP.NET`, `WPF`, or `WinForms`.

It's recommended that shared dependencies be loaded into AssemblyLoadContext.Default. This sharing is the common design pattern.

Sharing is implemented in the coding of the custom AssemblyLoadContext instance. AssemblyLoadContext has various events and virtual functions that can be overridden. When any of these functions return a reference to an Assembly instance that was loaded in another AssemblyLoadContext instance, the Assembly instance is shared. The standard load algorithm defers to AssemblyLoadContext.Default for loading to simplify the common sharing pattern. For more information, see Managed assembly loading algorithm.

Type-conversion issues

When two AssemblyLoadContext instances contain type definitions with the same `name`, they're not the same type. They're the same type if and only if they come from the same Assembly instance.

To complicate matters, exception messages about these mismatched types can be confusing. The types are referred to in the exception messages by their simple type names. The common exception message in this case is of the form:

> Object of type 'IsolatedType' cannot be converted to type 'IsolatedType'.

Debug type-conversion issues

Given a pair of mismatched types, it's important to also know:

- Each type's Type.Assembly.
- Each type's AssemblyLoadContext, which can be obtained via the AssemblyLoadContext.GetLoadContext(Assembly) function.

Given two objects `a` and `b`, evaluating the following in the debugger will be helpful:

C#

```
// In debugger look at each assembly's instance, Location, and FullName
a.GetType().Assembly
b.GetType().Assembly
// In debugger look at each AssemblyLoadContext's instance and name
System.Runtime.Loader.AssemblyLoadContext.GetLoadContext(a.GetType().Assembl
y)
System.Runtime.Loader.AssemblyLoadContext.GetLoadContext(b.GetType().Assembl
y)
```

Resolve type-conversion issues

There are two design patterns for solving these type conversion issues.

1. Use common shared types. This shared type can either be a primitive runtime type, or it can involve creating a new shared type in a shared assembly. Often the shared type is an interface defined in an application assembly. For more information, read about how dependencies are shared.

2. Use marshalling techniques to convert from one type to another.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Default probing

Article • 10/11/2022

The AssemblyLoadContext.Default instance is responsible for locating an assembly's dependencies. This article describes the AssemblyLoadContext.Default instance's probing logic.

Host configured probing properties

When the runtime is started, the runtime host provides a set of named probing properties that configure AssemblyLoadContext.Default probe paths.

Each probing property is optional. If present, each property is a string value that contains a delimited list of absolute paths. The delimiter is ';' on Windows and ':' on all other platforms.

Property Name	Description
`TRUSTED_PLATFORM_ASSEMBLIES`	List of platform and application assembly file paths.
`PLATFORM_RESOURCE_ROOTS`	List of directory paths to search for satellite resource assemblies.
`NATIVE_DLL_SEARCH_DIRECTORIES`	List of directory paths to search for unmanaged (native) libraries.
`APP_PATHS`	List of directory paths to search for managed assemblies.

How are the properties populated?

There are two main scenarios for populating the properties depending on whether the *<myapp>.deps.json* file exists.

- When the **.deps.json* file is present, it's parsed to populate the probing properties.
- When the **.deps.json* file isn't present, the application's directory is assumed to contain all the dependencies. The directory's contents are used to populate the probing properties.

Additionally, the **.deps.json* files for any referenced frameworks are similarly parsed.

The environment variable `DOTNET_ADDITIONAL_DEPS` can be used to add additional dependencies. `dotnet.exe` also contains an optional `--additional-deps` parameter to set this value on application startup.

The `APP_PATHS` property is not populated by default and is omitted for most applications.

The list of all **.deps.json* files used by the application can be accessed via `System.AppContext.GetData("APP_CONTEXT_DEPS_FILES")`.

How do I see the probing properties from managed code?

Each property is available by calling the AppContext.GetData(String) function with the property name from the table above.

How do I debug the probing properties' construction?

The .NET Core runtime host will output useful trace messages when certain environment variables are enabled:

Environment Variable	Description
`COREHOST_TRACE=1`	Enables tracing.
`COREHOST_TRACEFILE=<path>`	Traces to a file path instead of the default `stderr`.
`COREHOST_TRACE_VERBOSITY`	Sets the verbosity from 1 (lowest) to 4 (highest).

Managed assembly default probing

When probing to locate a managed assembly, the AssemblyLoadContext.Default looks in order at:

- Files matching the AssemblyName.Name in `TRUSTED_PLATFORM_ASSEMBLIES` (after removing file extensions).
- Assembly files in `APP_PATHS` with common file extensions.

Satellite (resource) assembly probing

To find a satellite assembly for a specific culture, construct a set of file paths.

For each path in `PLATFORM_RESOURCE_ROOTS` and then `APP_PATHS`, append the CultureInfo.Name string, a directory separator, the AssemblyName.Name string, and the extension '.dll'.

If any matching file exists, attempt to load and return it.

Unmanaged (native) library probing

The runtime's unmanaged library probing algorithm is identical on all platforms. However, since the actual load of the unmanaged library is performed by the underlying platform, the observed behavior can be slightly different.

1. Check if the supplied library name represents an absolute or relative path.
2. If the name represents an absolute path, use the name directly for all subsequent operations. Otherwise, use the name and create platform-defined combinations to consider. Combinations consist of platform specific prefixes (for example, `lib`) and/or suffixes (for example, `.dll`, `.dylib`, and `.so`). This is not an exhaustive list, and it doesn't represent the exact effort made on each platform. It's just an example of what is considered. For more information, see native library loading.
3. The name and, if the path is relative, each combination, is then used in the following steps. The first successful load attempt immediately returns the handle to the loaded library.
    - Append it to each path supplied in the `NATIVE_DLL_SEARCH_DIRECTORIES` property and attempt to load.
    - If DefaultDllImportSearchPathsAttribute is either not defined on the calling assembly or p/invoke or is defined and includes `DllImportSearchPath.AssemblyDirectory`, append the name or combination to the calling assembly's directory and attempt to load.
    - Use it directly to load the library.
4. Indicate that the library failed to load.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

Managed assembly loading algorithm

Article • 08/30/2022

Managed assemblies are located and loaded with an algorithm that has various stages.

All managed assemblies except satellite assemblies and `WinRT` assemblies use the same algorithm.

When are managed assemblies loaded?

The most common mechanism to trigger a managed assembly load is a static assembly reference. These references are inserted by the compiler whenever code uses a type defined in another assembly. These assemblies are loaded (`load-by-name`) as needed by the runtime. The exact timing of when the static assembly references are loaded is unspecified. It can vary between runtime versions and is influenced by optimizations like inlining.

The direct use of the following APIs will also trigger loads:

Expand table

API	Description	`Active` **AssemblyLoadContext**
AssemblyLoadContext.LoadFromAssemblyName	`Load-by-name`	The this instance.
AssemblyLoadContext.LoadFromAssemblyPath AssemblyLoadContext.LoadFromNativeImagePath	Load from path.	The this instance.
AssemblyLoadContext.LoadFromStream	Load from object.	The this instance.
Assembly.LoadFile	Load from path in a new AssemblyLoadContext instance	The new AssemblyLoadContext instance.
Assembly.LoadFrom	Load from path in the AssemblyLoadContext.Default instance. Adds an AppDomain.AssemblyResolve handler. The handler will load the assembly's dependencies from its directory.	The AssemblyLoadContext.Default instance.
Assembly.Load(AssemblyName) Assembly.Load(String) Assembly.LoadWithPartialName	`Load-by-name`.	Inferred from caller. Prefer AssemblyLoadContext methods.
Assembly.Load(Byte[]) Assembly.Load(Byte[], Byte[])	Load from object in a new AssemblyLoadContext	The new AssemblyLoadContext

API	Description	`Active` AssemblyLoadContext
	instance.	instance.
Type.GetType(String) Type.GetType(String, Boolean) Type.GetType(String, Boolean, Boolean)	`Load-by-name`.	Inferred from caller. Prefer Type.GetType methods with an `assemblyResolver` argument.
Assembly.GetType	If type `name` describes an assembly qualified generic type, trigger a `Load-by-name`.	Inferred from caller. Prefer Type.GetType when using assembly qualified type names.
Activator.CreateInstance(String, String) Activator.CreateInstance(String, String, Object[]) Activator.CreateInstance(String, String, Boolean, BindingFlags, Binder, Object[], CultureInfo, Object[])	`Load-by-name`.	Inferred from caller. Prefer Activator.CreateInstance methods taking a Type argument.

Algorithm

The following algorithm describes how the runtime loads a managed assembly.

1. Determine the `active` AssemblyLoadContext.

   - For a static assembly reference, the `active` AssemblyLoadContext is the instance that loaded the referring assembly.
   - Preferred APIs make the `active` AssemblyLoadContext explicit.
   - Other APIs infer the `active` AssemblyLoadContext. For these APIs, the AssemblyLoadContext.CurrentContextualReflectionContext property is used. If its value is `null`, then the inferred AssemblyLoadContext instance is used.
   - See the table in the When are managed assemblies loaded? section.

2. For the `Load-by-name` methods, the `active` AssemblyLoadContext loads the assembly in the following priority order:

   - Check its `cache-by-name`.
   - Call the AssemblyLoadContext.Load function.
   - Check the AssemblyLoadContext.Default instance's cache and run managed assembly default probing logic. If an assembly is newly loaded, a reference is added to the AssemblyLoadContext.Default instance's `cache-by-name`.
   - Raise the AssemblyLoadContext.Resolving event for the active AssemblyLoadContext.
   - Raise the AppDomain.AssemblyResolve event.

3. For the other types of loads, the `active` AssemblyLoadContext loads the assembly in the following priority order:

- Check its `cache-by-name`.
- Load from the specified path or raw assembly object. If an assembly is newly loaded, a reference is added to the `active` AssemblyLoadContext instance's `cache-by-name`.

4. In either case, if an assembly is newly loaded, then the AppDomain.AssemblyLoad event is raised.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Satellite assembly loading algorithm

Article • 11/11/2023

Satellite assemblies are used to store localized resources customized for language and culture.

Satellite assemblies use a different loading algorithm than general managed assemblies.

When are satellite assemblies loaded?

Satellite assemblies are loaded when loading a localized resource.

The basic API to load localized resources is the System.Resources.ResourceManager class. Ultimately the ResourceManager class will call the GetSatelliteAssembly method for each CultureInfo.Name.

Higher-level APIs may abstract the low-level API.

Algorithm

The .NET Core resource fallback process involves the following steps:

1. Determine the `active` AssemblyLoadContext instance. In all cases, the `active` instance is the executing assembly's AssemblyLoadContext.
2. The `active` instance loads a satellite assembly for the requested culture in the following priority order:
    - Check its cache.
    - If `active` is the AssemblyLoadContext.Default instance, run the default satellite (resource) assembly probing logic.
    - Call the AssemblyLoadContext.Load function.
    - If the managed assembly corresponding to the satellite assembly was loaded from a file, check the directory of the managed assembly for a subdirectory that matches the requested CultureInfo.Name (for example, `es-MX`).

      > **Note**
      >
      > On Linux and macOS, the subdirectory is case-sensitive and must either:

    - Exactly match case.
    - Be in lower case.

  - Raise the AssemblyLoadContext.Resolving event.
  - Raise the AppDomain.AssemblyResolve event.

3. If a satellite assembly is loaded:

   - The AppDomain.AssemblyLoad event is raised.
   - The assembly is searched for the requested resource. If the runtime finds the resource in the assembly, it uses it. If it doesn't find the resource, it continues the search.

   > **Note**
   >
   > To find a resource within the satellite assembly, the runtime searches for the resource file requested by the **ResourceManager** for the current **CultureInfo.Name**. Within the resource file, it searches for the requested resource name. If either is not found, the resource is treated as not found.

4. The ResourceManager next searches the parent culture assemblies through many potential levels, each time repeating steps 2 & 3.

   Each culture has only one parent, which is defined by the CultureInfo.Parent property.

   The search for parent cultures stops when a culture's Parent property is CultureInfo.InvariantCulture.

   For the InvariantCulture, we don't return to steps 2 & 3, but rather continue with step 5.

5. If the resource is still not found, the ResourceManager uses the resource for the default (fallback) culture.

   Typically, the resources for the default culture are included in the main application assembly. However, you can specify UltimateResourceFallbackLocation.Satellite for the NeutralResourcesLanguageAttribute.Location property. This value indicates that the ultimate fallback location for resources is a satellite assembly rather than the main assembly.

> **Note**
>
> The default culture is the ultimate fallback. Therefore, we recommend that you always include an exhaustive set of resources in the default resource file. This helps prevent exceptions from being thrown. By having an exhaustive set, you provide a fallback for all resources and ensure that at least one resource is always present for the user, even if it is not culturally specific.

6. Finally,

   - If the runtime doesn't find a resource file for a default (fallback) culture, a MissingManifestResourceException or MissingSatelliteAssemblyException exception is thrown.
   - If the resource file is found but the requested resource isn't present, the request returns `null`.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Unmanaged (native) library loading algorithm

Article • 10/27/2021

Unmanaged libraries are located and loaded with an algorithm involving various stages.

The following algorithm describes how native libraries are loaded through `PInvoke`.

`PInvoke` load library algorithm

`PInvoke` uses the following algorithm when attempting to load an unmanaged assembly:

1. Determine the `active` AssemblyLoadContext. For an unmanaged load library, the `active` AssemblyLoadContext is the one with the assembly that defines the `PInvoke`.

2. For the `active` AssemblyLoadContext, try to find the assembly in priority order by:

    - Checking its cache.
    - Calling the current System.Runtime.InteropServices.DllImportResolver delegate set by the NativeLibrary.SetDllImportResolver(Assembly, DllImportResolver) function.
    - Calling the AssemblyLoadContext.LoadUnmanagedDll function on the `active` AssemblyLoadContext.
    - Checking the AppDomain instance's cache and running the Unmanaged (native) library probing logic.
    - Raising the AssemblyLoadContext.ResolvingUnmanagedDll event for the `active` AssemblyLoadContext.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

issues and pull requests. For more information, see our contributor guide.

Provide product feedback

Collect detailed assembly loading information

Article • 11/08/2021

Starting with .NET 5, the runtime can emit events through `EventPipe` with detailed information about managed assembly loading to aid in diagnosing assembly loading issues. These events are emitted by the `Microsoft-Windows-DotNETRuntime` provider under the `AssemblyLoader` keyword (`0x4`).

Prerequisites

- .NET 5 SDK or later versions
- dotnet-trace tool

> **Note**
>
> The scope of `dotnet-trace` capabilities is greater than collecting detailed assembly loading information. For more information on the usage of `dotnet-trace`, see **dotnet-trace**.

Collect a trace with assembly loading events

You can use `dotnet-trace` to trace an existing process or to launch a child process and trace it from startup.

Trace an existing process

To enable assembly loading events in the runtime and collect a trace of them, use `dotnet-trace` with the following command:

```
Console
dotnet-trace collect --providers Microsoft-Windows-DotNETRuntime:4 --process-id <pid>
```

This command collects a trace of the specified `<pid>`, enabling the `AssemblyLoader` events in the `Microsoft-Windows-DotNETRuntime` provider. The result is a `.nettrace` file.

Use dotnet-trace to launch a child process and trace it from startup

Sometimes it may be useful to collect a trace of a process from its startup. For apps running .NET 5 or later, you can use `dotnet-trace` to do this.

The following command launches *hello.exe* with `arg1` and `arg2` as its command line arguments and collects a trace from its runtime startup:

```
Console
dotnet-trace collect --providers Microsoft-Windows-DotNETRuntime:4 -- hello.exe arg1 arg2
```

You can stop collecting the trace by pressing `Enter` or `Ctrl` + `C`. This also closes *hello.exe*.

> **Note**
>
> - Launching *hello.exe* via `dotnet-trace` redirects its input and output, and you won't be able to interact with it on the console by default. Use the `--show-child-io` switch to interact with its `stdin` and `stdout`.
> - Exiting the tool via `Ctrl`+`C` or `SIGTERM` safely ends both the tool and the child process.
> - If the child process exits before the tool, the tool exits as well and the trace should be safely viewable.

View a trace

The collected trace file can be viewed on Windows using the Events view in PerfView. All the assembly loading events will be prefixed with `Microsoft-Windows-DotNETRuntime/AssemblyLoader`.

Example (on Windows)

This example uses the assembly loading extension points sample. The application attempts to load an assembly `MyLibrary` - an assembly that is not referenced by the application and thus requires handling in an assembly loading extension point to be successfully loaded.

Collect the trace

1. Navigate to the directory with the downloaded sample. Build the application with:

   ```
   Console

   dotnet build
   ```

2. Launch the application with arguments indicating that it should pause, waiting for a key press. On resuming, it will attempt to load the assembly in the default `AssemblyLoadContext` - without the handling necessary for a successful load. Navigate to the output directory and run:

   ```
   Console

   AssemblyLoading.exe /d default
   ```

3. Find the application's process ID.

   ```
   Console

   dotnet-trace ps
   ```

   The output will list the available processes. For example:

   ```
   Console

   35832 AssemblyLoading C:\src\AssemblyLoading\bin\Debug\net5.0\AssemblyLoading.exe
   ```

4. Attach `dotnet-trace` to the running application.

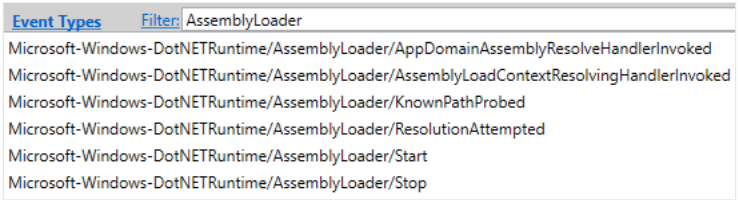
   ```
   Console

   dotnet-trace collect --providers Microsoft-Windows-DotNETRuntime:4 --process-id 35832
   ```

5. In the window running the application, press any key to let the program continue. Tracing will automatically stop once the application exits.

View the trace

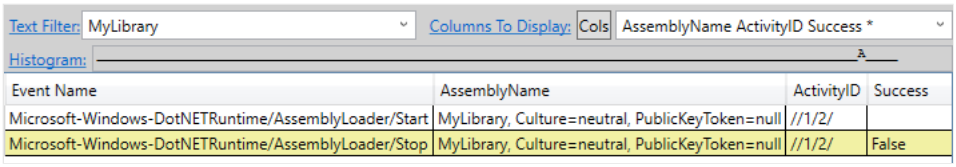
Open the collected trace in PerfView and open the Events view. Filter the events list to `Microsoft-Windows-DotNETRuntime/AssemblyLoader` events.



All assembly loads that occurred in the application after tracing started will be shown. To inspect the load operation for the assembly of interest for this example - `MyLibrary`, we can do some more filtering.

Assembly loads

Filter the view to the Start and Stop events under `Microsoft-Windows-DotNETRuntime/AssemblyLoader` using the event list on the left. Add the columns `AssemblyName`, `ActivityID`, and `Success` to the view. Filter to events containing `MyLibrary`.

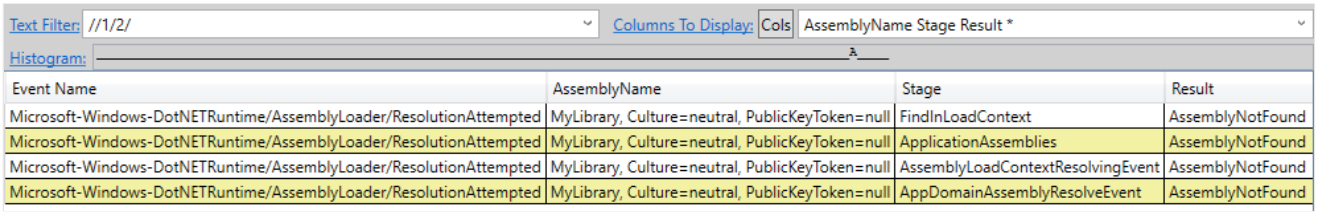


Event Name	AssemblyName	ActivityID	Success
`AssemblyLoader/Start`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/	
`AssemblyLoader/Stop`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/	False

You should see one `Start`/`Stop` pair with `Success=False` on the `Stop` event, indicating the load operation failed. Note that the two events have the same activity ID. The activity ID can be used to filter all the other assembly loader events to just the ones corresponding to this load operation.

Breakdown of attempt to load

For a more detailed breakdown of the load operation, filter the view to the ResolutionAttempted events under `Microsoft-Windows-DotNETRuntime/AssemblyLoader` using the event list on the left. Add the columns `AssemblyName`, `Stage`, and `Result` to the view. Filter to events with the activity ID from the `Start`/`Stop` pair.

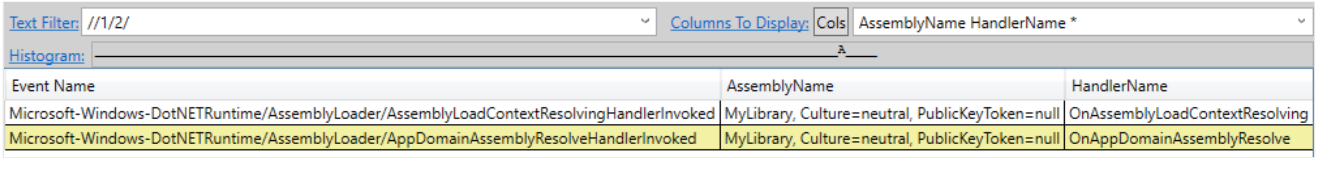


Event Name	AssemblyName	Stage	Result
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`FindInLoadContext`	`AssemblyNotFound`
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`ApplicationAssemblies`	`AssemblyNotFound`
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`AssemblyLoadContextResolvingEvent`	`AssemblyNotFound`
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`AppDomainAssemblyResolveEvent`	`AssemblyNotFound`

The events above indicate that the assembly loader attempted to resolve the assembly by looking in the current load context, running the default probing logic for managed application assemblies, invoking handlers for the AssemblyLoadContext.Resolving event, and invoking handlers for the AppDomain.AssemblyResolve. For all of these steps, the assembly was not found.

Extension points

To see which extension points were invoked, filter the view to the AssemblyLoadContextResolvingHandlerInvoked and AppDomainAssemblyResolveHandlerInvoked under `Microsoft-Windows-DotNETRuntime/AssemblyLoader` using the event list on the left. Add the columns `AssemblyName` and `HandlerName` to the view. Filter to events with the activity ID from the `Start`/`Stop` pair.



Event Name	AssemblyName	HandlerName
`AssemblyLoader/AssemblyLoadContextResolvingHandlerInvoked`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`OnAssemblyLoadContextResolving`
`AssemblyLoader/AppDomainAssemblyResolveHandlerInvoked`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`OnAppDomainAssemblyResolve`

The events above indicate that a handler named `OnAssemblyLoadContextResolving` was invoked for the AssemblyLoadContext.Resolving event and a handler named `OnAppDomainAssemblyResolve` was invoked for the AppDomain.AssemblyResolve event.

Collect another trace

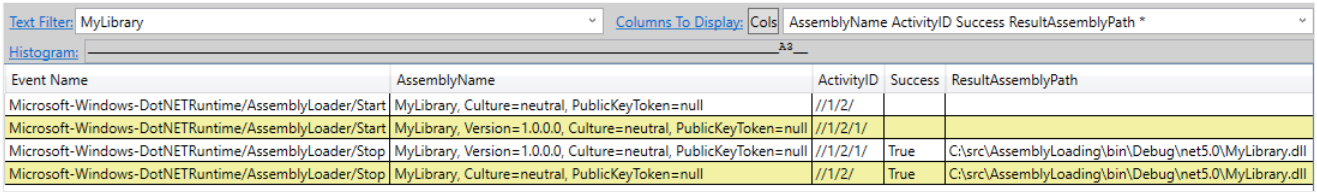
Run the application with arguments such that its handler for the AssemblyLoadContext.Resolving event will load the `MyLibrary` assembly.

```
Console

AssemblyLoading /d default alc-resolving
```

Collect and open another `.nettrace` file using the steps from above.

Filter to the `Start` and `Stop` events for `MyLibrary` again. You should see a `Start`/`Stop` pair with another `Start`/`Stop` between them. The inner load operation represents the load triggered by the handler for AssemblyLoadContext.Resolving when it called AssemblyLoadContext.LoadFromAssemblyPath. This time, you should see `Success=True` on the `Stop` event, indicating the load operation succeeded. The `ResultAssemblyPath` field shows the path of the resulting assembly.

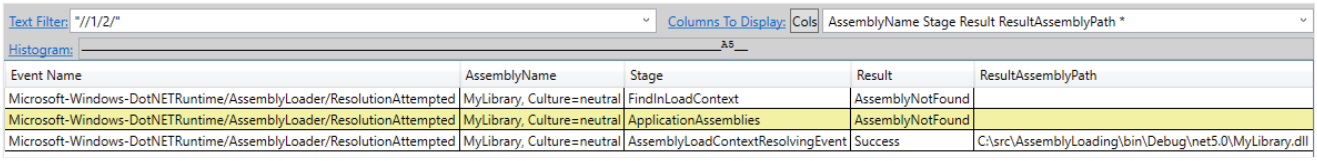

Text Filter: MyLibrary | Columns To Display: Cols | AssemblyName ActivityID Success ResultAssemblyPath *

Histogram:

Event Name	AssemblyName	ActivityID	Success	ResultAssemblyPath
Microsoft-Windows-DotNETRuntime/AssemblyLoader/Start	MyLibrary, Culture=neutral, PublicKeyToken=null	//1/2/		
Microsoft-Windows-DotNETRuntime/AssemblyLoader/Start	MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null	//1/2/1/		
Microsoft-Windows-DotNETRuntime/AssemblyLoader/Stop	MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null	//1/2/1/	True	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll
Microsoft-Windows-DotNETRuntime/AssemblyLoader/Stop	MyLibrary, Culture=neutral, PublicKeyToken=null	//1/2/	True	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll

Event Name	AssemblyName	ActivityID	Success	ResultAssemblyPath
`AssemblyLoader/Start`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/		
`AssemblyLoader/Start`	`MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`	//1/2/1/		
`AssemblyLoader/Stop`	`MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`	//1/2/1/	True	*C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll*
`AssemblyLoader/Stop`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/	True	*C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll*

We can then look at the `ResolutionAttempted` events with the activity ID from the outer load to determine the step at which the assembly was successfully resolved. This time, the events will show that the `AssemblyLoadContextResolvingEvent` stage was successful. The `ResultAssemblyPath` field shows the path of the resulting assembly.

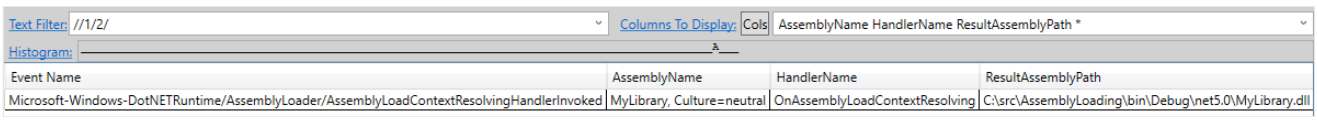

Text Filter: "//1/2/" | Columns To Display: Cols | AssemblyName Stage Result ResultAssemblyPath *

Histogram:

Event Name	AssemblyName	Stage	Result	ResultAssemblyPath
Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted	MyLibrary, Culture=neutral	FindInLoadContext	AssemblyNotFound	
Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted	MyLibrary, Culture=neutral	ApplicationAssemblies	AssemblyNotFound	
Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted	MyLibrary, Culture=neutral	AssemblyLoadContextResolvingEvent	Success	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll

Event Name	AssemblyName	Stage	Result	ResultAssemblyPath
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`FindInLoadContext`	`AssemblyNotFound`	
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`ApplicationAssemblies`	`AssemblyNotFound`	
`AssemblyLoader/ResolutionAttempted`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`AssemblyLoadContextResolvingEvent`	`Success`	*C:\src\AssemblyLoading\bin\Debug\net5.0\My*

Looking at `AssemblyLoadContextResolvingHandlerInvoked` events will show that the handler named `OnAssemblyLoadContextResolving` was invoked. The `ResultAssemblyPath` field shows the path of the assembly returned by the handler.


Text Filter: //1/2/ | Columns To Display: Cols | AssemblyName HandlerName ResultAssemblyPath *

Histogram:

Event Name	AssemblyName	HandlerName	ResultAssemblyPath
Microsoft-Windows-DotNETRuntime/AssemblyLoader/AssemblyLoadContextResolvingHandlerInvoked	MyLibrary, Culture=neutral	OnAssemblyLoadContextResolving	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll

Event Name	AssemblyName	HandlerName	ResultAssemblyPath
`AssemblyLoader/AssemblyLoadContextResolvingHandlerInvoked`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`OnAssemblyLoadContextResolving`	*C:\src\AssemblyLoading\bin\Debug\net5.0\M*

Note that there is no longer a `ResolutionAttempted` event with the `AppDomainAssemblyResolveEvent` stage or any `AppDomainAssemblyResolveHandlerInvoked` events, as the assembly was successfully loaded before reaching the step of the loading algorithm that raises the AppDomain.AssemblyResolve event.

See also

- Assembly loader events
- dotnet-trace
- PerfView

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Create a .NET Core application with plugins

Article • 02/04/2022

This tutorial shows you how to create a custom AssemblyLoadContext to load plugins. An AssemblyDependencyResolver is used to resolve the dependencies of the plugin. The tutorial correctly isolates the plugin's dependencies from the hosting application. You'll learn how to:

- Structure a project to support plugins.
- Create a custom AssemblyLoadContext to load each plugin.
- Use the System.Runtime.Loader.AssemblyDependencyResolver type to allow plugins to have dependencies.
- Author plugins that can be easily deployed by just copying the build artifacts.

Prerequisites

- Install the .NET 5 SDK or a newer version.

> **Note**
>
> The sample code targets .NET 5, but all the features it uses were introduced in .NET Core 3.0 and are available in all .NET releases since then.

Create the application

The first step is to create the application:

1. Create a new folder, and in that folder run the following command:

   .NET CLI

   ```
   dotnet new console -o AppWithPlugin
   ```

2. To make building the project easier, create a Visual Studio solution file in the same folder. Run the following command:

   .NET CLI

```
dotnet new sln
```

3. Run the following command to add the app project to the solution:

.NET CLI

```
dotnet sln add AppWithPlugin/AppWithPlugin.csproj
```

Now we can fill in the skeleton of our application. Replace the code in the *AppWithPlugin/Program.cs* file with the following code:

C#

```
using PluginBase;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace AppWithPlugin
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                if (args.Length == 1 && args[0] == "/d")
                {
                    Console.WriteLine("Waiting for any key...");
                    Console.ReadLine();
                }

                // Load commands from plugins.

                if (args.Length == 0)
                {
                    Console.WriteLine("Commands: ");
                    // Output the loaded commands.
                }
                else
                {
                    foreach (string commandName in args)
                    {
                        Console.WriteLine($"-- {commandName} --");

                        // Execute the command with the name passed as an
argument.
```

```
                        Console.WriteLine();
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```

Create the plugin interfaces

The next step in building an app with plugins is defining the interface the plugins need to implement. We suggest that you make a class library that contains any types that you plan to use for communicating between your app and plugins. This division allows you to publish your plugin interface as a package without having to ship your full application.

In the root folder of the project, run `dotnet new classlib -o PluginBase`. Also, run `dotnet sln add PluginBase/PluginBase.csproj` to add the project to the solution file. Delete the `PluginBase/Class1.cs` file, and create a new file in the `PluginBase` folder named `ICommand.cs` with the following interface definition:

C#

```
namespace PluginBase
{
    public interface ICommand
    {
        string Name { get; }
        string Description { get; }

        int Execute();
    }
}
```

This `ICommand` interface is the interface that all of the plugins will implement.

Now that the `ICommand` interface is defined, the application project can be filled in a little more. Add a reference from the `AppWithPlugin` project to the `PluginBase` project with the `dotnet add AppWithPlugin/AppWithPlugin.csproj reference PluginBase/PluginBase.csproj` command from the root folder.

Replace the `// Load commands from plugins` comment with the following code snippet to enable it to load plugins from given file paths:

```
C#

string[] pluginPaths = new string[]
{
    // Paths to plugins to load.
};

IEnumerable<ICommand> commands = pluginPaths.SelectMany(pluginPath =>
{
    Assembly pluginAssembly = LoadPlugin(pluginPath);
    return CreateCommands(pluginAssembly);
}).ToList();
```

Then replace the `// Output the loaded commands` comment with the following code snippet:

```
C#

foreach (ICommand command in commands)
{
    Console.WriteLine($"{command.Name}\t - {command.Description}");
}
```

Replace the `// Execute the command with the name passed as an argument` comment with the following snippet:

```
C#

ICommand command = commands.FirstOrDefault(c => c.Name == commandName);
if (command == null)
{
    Console.WriteLine("No such command is known.");
    return;
}

command.Execute();
```

And finally, add static methods to the `Program` class named `LoadPlugin` and `CreateCommands`, as shown here:

```
C#

static Assembly LoadPlugin(string relativePath)
{
```

```
    throw new NotImplementedException();
}

static IEnumerable<ICommand> CreateCommands(Assembly assembly)
{
    int count = 0;

    foreach (Type type in assembly.GetTypes())
    {
        if (typeof(ICommand).IsAssignableFrom(type))
        {
            ICommand result = Activator.CreateInstance(type) as ICommand;
            if (result != null)
            {
                count++;
                yield return result;
            }
        }
    }

    if (count == 0)
    {
        string availableTypes = string.Join(",",
assembly.GetTypes().Select(t => t.FullName));
        throw new ApplicationException(
            $"Can't find any type which implements ICommand in {assembly}
from {assembly.Location}.\n" +
            $"Available types: {availableTypes}");
    }
}
```

Load plugins

Now the application can correctly load and instantiate commands from loaded plugin assemblies, but it's still unable to load the plugin assemblies. Create a file named *PluginLoadContext.cs* in the *AppWithPlugin* folder with the following contents:

C#

```
using System;
using System.Reflection;
using System.Runtime.Loader;

namespace AppWithPlugin
{
    class PluginLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public PluginLoadContext(string pluginPath)
        {
```

```
            _resolver = new AssemblyDependencyResolver(pluginPath);
        }

        protected override Assembly Load(AssemblyName assemblyName)
        {
            string assemblyPath =
_resolver.ResolveAssemblyToPath(assemblyName);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }

        protected override IntPtr LoadUnmanagedDll(string unmanagedDllName)
        {
            string libraryPath =
_resolver.ResolveUnmanagedDllToPath(unmanagedDllName);
            if (libraryPath != null)
            {
                return LoadUnmanagedDllFromPath(libraryPath);
            }

            return IntPtr.Zero;
        }
    }
}
```

The `PluginLoadContext` type derives from AssemblyLoadContext. The `AssemblyLoadContext` type is a special type in the runtime that allows developers to isolate loaded assemblies into different groups to ensure that assembly versions don't conflict. Additionally, a custom `AssemblyLoadContext` can choose different paths to load assemblies from and override the default behavior. The `PluginLoadContext` uses an instance of the `AssemblyDependencyResolver` type introduced in .NET Core 3.0 to resolve assembly names to paths. The `AssemblyDependencyResolver` object is constructed with the path to a .NET class library. It resolves assemblies and native libraries to their relative paths based on the *.deps.json* file for the class library whose path was passed to the `AssemblyDependencyResolver` constructor. The custom `AssemblyLoadContext` enables plugins to have their own dependencies, and the `AssemblyDependencyResolver` makes it easy to correctly load the dependencies.

Now that the `AppWithPlugin` project has the `PluginLoadContext` type, update the `Program.LoadPlugin` method with the following body:

```
C#
```

```
static Assembly LoadPlugin(string relativePath)
{
    // Navigate up to the solution root
    string root = Path.GetFullPath(Path.Combine(
        Path.GetDirectoryName(
            Path.GetDirectoryName(
                Path.GetDirectoryName(
                    Path.GetDirectoryName(

Path.GetDirectoryName(typeof(Program).Assembly.Location)))))));

    string pluginLocation = Path.GetFullPath(Path.Combine(root,
relativePath.Replace('\\', Path.DirectorySeparatorChar)));
    Console.WriteLine($"Loading commands from: {pluginLocation}");
    PluginLoadContext loadContext = new PluginLoadContext(pluginLocation);
    return loadContext.LoadFromAssemblyName(new
AssemblyName(Path.GetFileNameWithoutExtension(pluginLocation)));
}
```

By using a different `PluginLoadContext` instance for each plugin, the plugins can have different or even conflicting dependencies without issue.

Simple plugin with no dependencies

Back in the root folder, do the following:

1. Run the following command to create a new class library project named `HelloPlugin`:

   .NET CLI

   ```
   dotnet new classlib -o HelloPlugin
   ```

2. Run the following command to add the project to the `AppWithPlugin` solution:

   .NET CLI

   ```
   dotnet sln add HelloPlugin/HelloPlugin.csproj
   ```

3. Replace the *HelloPlugin/Class1.cs* file with a file named *HelloCommand.cs* with the following contents:

C#

```
using PluginBase;
using System;
```

```
namespace HelloPlugin
{
    public class HelloCommand : ICommand
    {
        public string Name { get => "hello"; }
        public string Description { get => "Displays hello message."; }

        public int Execute()
        {
            Console.WriteLine("Hello !!!");
            return 0;
        }
    }
}
```

Now, open the *HelloPlugin.csproj* file. It should look similar to the following:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

</Project>
```

In between the `<PropertyGroup>` tags, add the following element:

XML

```
  <EnableDynamicLoading>true</EnableDynamicLoading>
```

The `<EnableDynamicLoading>true</EnableDynamicLoading>` prepares the project so that it can be used as a plugin. Among other things, this will copy all of its dependencies to the output of the project. For more details see EnableDynamicLoading.

In between the `<Project>` tags, add the following elements:

XML

```
<ItemGroup>
    <ProjectReference Include="..\PluginBase\PluginBase.csproj">
        <Private>false</Private>
        <ExcludeAssets>runtime</ExcludeAssets>
```

```
    </ProjectReference>
  </ItemGroup>
```

The `<Private>false</Private>` element is important. This tells MSBuild to not copy *PluginBase.dll* to the output directory for HelloPlugin. If the *PluginBase.dll* assembly is present in the output directory, `PluginLoadContext` will find the assembly there and load it when it loads the *HelloPlugin.dll* assembly. At this point, the `HelloPlugin.HelloCommand` type will implement the `ICommand` interface from the *PluginBase.dll* in the output directory of the `HelloPlugin` project, not the `ICommand` interface that is loaded into the default load context. Since the runtime sees these two types as different types from different assemblies, the `AppWithPlugin.Program.CreateCommands` method won't find the commands. As a result, the `<Private>false</Private>` metadata is required for the reference to the assembly containing the plugin interfaces.

Similarly, the `<ExcludeAssets>runtime</ExcludeAssets>` element is also important if the `PluginBase` references other packages. This setting has the same effect as `<Private>false</Private>` but works on package references that the `PluginBase` project or one of its dependencies may include.

Now that the `HelloPlugin` project is complete, you should update the `AppWithPlugin` project to know where the `HelloPlugin` plugin can be found. After the `// Paths to plugins to load` comment, add `@"HelloPlugin\bin\Debug\net5.0\HelloPlugin.dll"` (this path could be different based on the .NET Core version you use) as an element of the `pluginPaths` array.

Plugin with library dependencies

Almost all plugins are more complex than a simple "Hello World", and many plugins have dependencies on other libraries. The `JsonPlugin` and `OldJsonPlugin` projects in the sample show two examples of plugins with NuGet package dependencies on `Newtonsoft.Json`. Because of this, all plugin projects should add `<EnableDynamicLoading>true</EnableDynamicLoading>` to the project properties so that they copy all of their dependencies to the output of `dotnet build`. Publishing the class library with `dotnet publish` will also copy all of its dependencies to the publish output.

Other examples in the sample

The complete source code for this tutorial can be found in the dotnet/samples repository. The completed sample includes a few other examples of

`AssemblyDependencyResolver` behavior. For example, the `AssemblyDependencyResolver` object can also resolve native libraries as well as localized satellite assemblies included in NuGet packages. The `UVPlugin` and `FrenchPlugin` in the samples repository demonstrate these scenarios.

Reference a plugin interface from a NuGet package

Let's say that there is an app A that has a plugin interface defined in the NuGet package named `A.PluginBase`. How do you reference the package correctly in your plugin project? For project references, using the `<Private>false</Private>` metadata on the `ProjectReference` element in the project file prevented the dll from being copied to the output.

To correctly reference the `A.PluginBase` package, you want to change the `<PackageReference>` element in the project file to the following:

XML

```
<PackageReference Include="A.PluginBase" Version="1.0.0">
    <ExcludeAssets>runtime</ExcludeAssets>
</PackageReference>
```

This prevents the `A.PluginBase` assemblies from being copied to the output directory of your plugin and ensures that your plugin will use A's version of `A.PluginBase`.

Plugin target framework recommendations

Because plugin dependency loading uses the *.deps.json* file, there is a gotcha related to the plugin's target framework. Specifically, your plugins should target a runtime, such as .NET 5, instead of a version of .NET Standard. The *.deps.json* file is generated based on which framework the project targets, and since many .NET Standard-compatible packages ship reference assemblies for building against .NET Standard and implementation assemblies for specific runtimes, the *.deps.json* may not correctly see implementation assemblies, or it may grab the .NET Standard version of an assembly instead of the .NET Core version you expect.

Plugin framework references

Currently, plugins can't introduce new frameworks into the process. For example, you can't load a plugin that uses the `Microsoft.AspNetCore.App` framework into an application that only uses the root `Microsoft.NETCore.App` framework. The host application must declare references to all frameworks needed by plugins.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to use and debug assembly unloadability in .NET

Article • 11/13/2023

.NET (Core) introduced the ability to load and later unload a set of assemblies. In .NET Framework, custom app domains were used for this purpose, but .NET (Core) only supports a single default app domain.

Unloadability is supported through AssemblyLoadContext. You can load a set of assemblies into a collectible `AssemblyLoadContext`, execute methods in them or just inspect them using reflection, and finally unload the `AssemblyLoadContext`. That unloads the assemblies loaded into the `AssemblyLoadContext`.

There's one noteworthy difference between the unloading using `AssemblyLoadContext` and using AppDomains. With AppDomains, the unloading is forced. At unload time, all threads running in the target AppDomain are aborted, managed COM objects created in the target AppDomain are destroyed, and so on. With `AssemblyLoadContext`, the unload is "cooperative". Calling the AssemblyLoadContext.Unload method just initiates the unloading. The unloading finishes after:

- No threads have methods from the assemblies loaded into the `AssemblyLoadContext` on their call stacks.
- None of the types from the assemblies loaded into the `AssemblyLoadContext`, instances of those types, and the assemblies themselves are referenced by:
  - References outside of the `AssemblyLoadContext`, except for weak references (WeakReference or WeakReference<T>).
  - Strong garbage collector (GC) handles (GCHandleType.Normal or GCHandleType.Pinned) from both inside and outside of the `AssemblyLoadContext`.

Use collectible AssemblyLoadContext

This section contains a detailed step-by-step tutorial that shows a simple way to load a .NET (Core) application into a collectible `AssemblyLoadContext`, execute its entry point, and then unload it. You can find a complete sample at https://github.com/dotnet/samples/tree/main/core/tutorials/Unloading.

Create a collectible AssemblyLoadContext

Derive your class from the AssemblyLoadContext and override its AssemblyLoadContext.Load method. That method resolves references to all assemblies that are dependencies of assemblies loaded into that `AssemblyLoadContext`.

The following code is an example of the simplest custom `AssemblyLoadContext`:

C#

```
class TestAssemblyLoadContext : AssemblyLoadContext
{
    public TestAssemblyLoadContext() : base(isCollectible: true)
    {
    }

    protected override Assembly? Load(AssemblyName name)
    {
        return null;
    }
}
```

As you can see, the `Load` method returns `null`. That means that all the dependency assemblies are loaded into the default context, and the new context contains only the assemblies explicitly loaded into it.

If you want to load some or all of the dependencies into the `AssemblyLoadContext` too, you can use the `AssemblyDependencyResolver` in the `Load` method. The `AssemblyDependencyResolver` resolves the assembly names to absolute assembly file paths. The resolver uses the *.deps.json* file and assembly files in the directory of the main assembly loaded into the context.

C#

```
using System.Reflection;
using System.Runtime.Loader;

namespace complex
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public TestAssemblyLoadContext(string mainAssemblyToLoadPath) :
base(isCollectible: true)
        {
            _resolver = new
AssemblyDependencyResolver(mainAssemblyToLoadPath);
        }

        protected override Assembly? Load(AssemblyName name)
```

```
        {
            string? assemblyPath = _resolver.ResolveAssemblyToPath(name);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }
    }
}
```

Use a custom collectible AssemblyLoadContext

This section assumes the simpler version of the `TestAssemblyLoadContext` is being used.

You can create an instance of the custom `AssemblyLoadContext` and load an assembly into it as follows:

C#

```
var alc = new TestAssemblyLoadContext();
Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
```

For each of the assemblies referenced by the loaded assembly, the `TestAssemblyLoadContext.Load` method is called so that the `TestAssemblyLoadContext` can decide where to get the assembly from. In this case, it returns `null` to indicate that it should be loaded into the default context from locations that the runtime uses to load assemblies by default.

Now that an assembly was loaded, you can execute a method from it. Run the `Main` method:

C#

```
var args = new object[1] {new string[] {"Hello"}};
_ = a.EntryPoint?.Invoke(null, args);
```

After the `Main` method returns, you can initiate unloading by either calling the `Unload` method on the custom `AssemblyLoadContext` or removing the reference you have to the `AssemblyLoadContext`:

C#

```
alc.Unload();
```

This is sufficient to unload the test assembly. Next, you'll put all of this into a separate noninlineable method to ensure that the `TestAssemblyLoadContext`, `Assembly`, and `MethodInfo` (the `Assembly.EntryPoint`) can't be kept alive by stack slot references (real- or JIT-introduced locals). That could keep the `TestAssemblyLoadContext` alive and prevent the unload.

Also, return a weak reference to the `AssemblyLoadContext` so that you can use it later to detect unload completion.

C#

```
[MethodImpl(MethodImplOptions.NoInlining)]
static void ExecuteAndUnload(string assemblyPath, out WeakReference
alcWeakRef)
{
    var alc = new TestAssemblyLoadContext();
    Assembly a = alc.LoadFromAssemblyPath(assemblyPath);

    alcWeakRef = new WeakReference(alc, trackResurrection: true);

    var args = new object[1] {new string[] {"Hello"}};
    _ = a.EntryPoint?.Invoke(null, args);

    alc.Unload();
}
```

Now you can run this function to load, execute, and unload the assembly.

C#

```
WeakReference testAlcWeakRef;
ExecuteAndUnload("absolute/path/to/your/assembly", out testAlcWeakRef);
```

However, the unload doesn't complete immediately. As previously mentioned, it relies on the garbage collector to collect all the objects from the test assembly. In many cases, it isn't necessary to wait for the unload completion. However, there are cases where it's useful to know that the unload has finished. For example, you might want to delete the assembly file that was loaded into the custom `AssemblyLoadContext` from disk. In such a case, the following code snippet can be used. It triggers garbage collection and waits for pending finalizers in a loop until the weak reference to the custom `AssemblyLoadContext` is set to `null`, indicating the target object was collected. In most cases, just one pass through the loop is required. However, for more complex cases where objects created

by the code running in the `AssemblyLoadContext` have finalizers, more passes might be needed.

```
C#

for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
{
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

The Unloading event

In some cases, it might be necessary for the code loaded into a custom `AssemblyLoadContext` to perform some cleanup when the unloading is initiated. For example, it might need to stop threads or clean up strong GC handles. The `Unloading` event can be used in such cases. You can hook a handler that performs the necessary cleanup to this event.

Troubleshoot unloadability issues

Due to the cooperative nature of the unloading, it's easy to forget about references that might be keeping the stuff in a collectible `AssemblyLoadContext` alive and preventing unload. Here's a summary of entities (some of them nonobvious) that can hold the references:

- Regular references held from outside of the collectible `AssemblyLoadContext` that are stored in a stack slot or a processor register (method locals, either explicitly created by the user code or implicitly by the just-in-time (JIT) compiler), a static variable, or a strong (pinning) GC handle, and transitively pointing to:
  - An assembly loaded into the collectible `AssemblyLoadContext`.
  - A type from such an assembly.
  - An instance of a type from such an assembly.
- Threads running code from an assembly loaded into the collectible `AssemblyLoadContext`.
- Instances of custom, noncollectible `AssemblyLoadContext` types created inside of the collectible `AssemblyLoadContext`.
- Pending RegisteredWaitHandle instances with callbacks set to methods in the custom `AssemblyLoadContext`.

Tip

Object references that are stored in stack slots or processor registers and that could prevent unloading of an `AssemblyLoadContext` can occur in the following situations:

- When function call results are passed directly to another function, even though there is no user-created local variable.
- When the JIT compiler keeps a reference to an object that was available at some point in a method.

Debug unloading issues

Debugging issues with unloading can be tedious. You can get into situations where you don't know what can be holding an `AssemblyLoadContext` alive, but the unload fails. The best tool to help with that is WinDbg (or LLDB on Unix) with the SOS plugin. You need to find what's keeping a `LoaderAllocator` that belongs to the specific `AssemblyLoadContext` alive. The SOS plugin lets you look at GC heap objects, their hierarchies, and roots.

To load the SOS plugin into the debugger, enter one of the following commands in the debugger command line.

In WinDbg (if it's not already loaded):

Console

```
.loadby sos coreclr
```

In LLDB:

Console

```
plugin load /path/to/libsosplugin.so
```

Now you'll debug an example program that has problems with unloading. The source code is available in the Example source code section. When you run it under WinDbg, the program breaks into the debugger right after attempting to check for the unload success. You can then start looking for the culprits.

Tip

If you debug using LLDB on Unix, the SOS commands in the following examples don't have the `!` in front of them.

Console

```
!dumpheap -type LoaderAllocator
```

This command dumps all objects with a type name containing `LoaderAllocator` that are in the GC heap. Here's an example:

Console

```
         Address               MT     Size
000002b78000ce40 00007ffadc93a288       48
000002b78000ceb0 00007ffadc93a218       24

Statistics:
              MT    Count    TotalSize Class Name
00007ffadc93a218        1           24
System.Reflection.LoaderAllocatorScout
00007ffadc93a288        1           48 System.Reflection.LoaderAllocator
Total 2 objects
```

In the "Statistics:" part, check the `MT` (`MethodTable`) that belongs to the `System.Reflection.LoaderAllocator`, which is the object you care about. Then, in the list at the beginning, find the entry with `MT` that matches that one, and get the address of the object itself. In this case, it's "000002b78000ce40".

Now that you know the address of the `LoaderAllocator` object, you can use another command to find its GC roots:

Console

```
!gcroot 0x000002b78000ce40
```

This command dumps the chain of object references that lead to the `LoaderAllocator` instance. The list starts with the root, which is the entity that keeps the `LoaderAllocator` alive and thus is the core of the problem. The root can be a stack slot, a processor register, a GC handle, or a static variable.

Here's an example of the output of the `gcroot` command:

Console

```
Thread 4ac:
    000000cf9499dd20 00007ffa7d0236bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
        rbp-20: 000000cf9499dd90
            ->  000002b78000d328 System.Reflection.RuntimeMethodInfo
            ->  000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
            ->  000002b78000d1d0 System.RuntimeType
            ->  000002b78000ce40 System.Reflection.LoaderAllocator

HandleTable:
    000002b7f8a81198 (strong handle)
    -> 000002b78000d948 test.Test
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

    000002b7f8a815f8 (pinned handle)
    -> 000002b790001038 System.Object[]
    -> 000002b78000d390 example.TestInfo
    -> 000002b78000d328 System.Reflection.RuntimeMethodInfo
    -> 000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
    -> 000002b78000d1d0 System.RuntimeType
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

Found 3 roots.
```

The next step is to figure out where the root is located so you can fix it. The easiest case is when the root is a stack slot or a processor register. In that case, the `gcroot` shows the name of the function whose frame contains the root and the thread executing that function. The difficult case is when the root is a static variable or a GC handle.

In the previous example, the first root is a local of type `System.Reflection.RuntimeMethodInfo` stored in the frame of the function `example.Program.Main(System.String[])` at address `rbp-20` (`rbp` is the processor register `rbp` and -20 is a hexadecimal offset from that register).

The second root is a normal (strong) `GCHandle` that holds a reference to an instance of the `test.Test` class.

The third root is a pinned `GCHandle`. This one is actually a static variable, but unfortunately, there's no way to tell. Statics for reference types are stored in a managed object array in internal runtime structures.

Another case that can prevent unloading of an `AssemblyLoadContext` is when a thread has a frame of a method from an assembly loaded into the `AssemblyLoadContext` on its stack. You can check that by dumping managed call stacks of all threads:

Console

```
~*e !clrstack
```

The command means "apply to all threads the `!clrstack` command". The following is the output of that command for the example. Unfortunately, LLDB on Unix doesn't have any way to apply a command to all threads, so you must manually switch threads and repeat the `clrstack` command. Ignore all threads where the debugger says "Unable to walk the managed stack".

Console

```
OS Thread Id: 0x6ba8 (0)
        Child SP               IP Call Site
0000001fc697d5c8 00007ffb50d9de12 [HelperMethodFrame: 0000001fc697d5c8]
System.Diagnostics.Debugger.BreakInternal()
0000001fc697d6d0 00007ffa864765fa System.Diagnostics.Debugger.Break()
0000001fc697d700 00007ffa864736bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
0000001fc697d998 00007ffae5fdc1e3 [GCFrame: 0000001fc697d998]
0000001fc697df28 00007ffae5fdc1e3 [GCFrame: 0000001fc697df28]
OS Thread Id: 0x2ae4 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x61a4 (2)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x7fdc (3)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5390 (4)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5ec8 (5)
        Child SP               IP Call Site
0000001fc70ff6e0 00007ffb5437f6e4 [DebuggerU2MCatchHandlerFrame:
0000001fc70ff6e0]
OS Thread Id: 0x4624 (6)
        Child SP               IP Call Site
GetFrameContext failed: 1
0000000000000000 0000000000000000
OS Thread Id: 0x60bc (7)
        Child SP               IP Call Site
0000001fc727f158 00007ffb5437fce4 [HelperMethodFrame: 0000001fc727f158]
System.Threading.Thread.SleepInternal(Int32)
```

```
0000001fc727f260 00007ffb37ea7c2b System.Threading.Thread.Sleep(Int32)
0000001fc727f290 00007ffa865005b3 test.Program.ThreadProc()
[E:\unloadability\test\Program.cs @ 17]
0000001fc727f2c0 00007ffb37ea6a5b
System.Threading.Thread.ThreadMain_ThreadStart()
0000001fc727f2f0 00007ffadbc4cbe3
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionCont
ext, System.Threading.ContextCallback, System.Object)
0000001fc727f568 00007ffae5fdc1e3 [GCFrame: 0000001fc727f568]
0000001fc727f7f0 00007ffae5fdc1e3 [DebuggerU2MCatchHandlerFrame:
0000001fc727f7f0]
```

As you can see, the last thread has `test.Program.ThreadProc()`. This is a function from the assembly loaded into the `AssemblyLoadContext`, and so it keeps the `AssemblyLoadContext` alive.

Example source code

The following code that contains unloadability issues is used in the previous debugging example.

Main testing program

C#

```
using System;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.Loader;

namespace example
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        public TestAssemblyLoadContext() : base(true)
        {
        }
        protected override Assembly? Load(AssemblyName name)
        {
            return null;
        }
    }

    class TestInfo
    {
        public TestInfo(MethodInfo? mi)
        {
            _entryPoint = mi;
        }
```

```
        MethodInfo? _entryPoint;
    }

    class Program
    {
        static TestInfo? entryPoint;

        [MethodImpl(MethodImplOptions.NoInlining)]
        static int ExecuteAndUnload(string assemblyPath, out WeakReference
testAlcWeakRef, out MethodInfo? testEntryPoint)
        {
            var alc = new TestAssemblyLoadContext();
            testAlcWeakRef = new WeakReference(alc);

            Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
            if (a == null)
            {
                testEntryPoint = null;
                Console.WriteLine("Loading the test assembly failed");
                return -1;
            }

            var args = new object[1] {new string[] {"Hello"}};

            // Issue preventing unloading #1 - we keep MethodInfo of a
method
            // for an assembly loaded into the TestAssemblyLoadContext in a
static variable.
            entryPoint = new TestInfo(a.EntryPoint);
            testEntryPoint = a.EntryPoint;

            var oResult = a.EntryPoint?.Invoke(null, args);
            alc.Unload();
            return (oResult is int result) ? result : -1;
        }

        static void Main(string[] args)
        {
            WeakReference testAlcWeakRef;
            // Issue preventing unloading #2 - we keep MethodInfo of a
method for an assembly loaded into the TestAssemblyLoadContext in a local
variable
            MethodInfo? testEntryPoint;
            int result = ExecuteAndUnload(@"absolute/path/to/test.dll", out
testAlcWeakRef, out testEntryPoint);

            for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
            {
                GC.Collect();
                GC.WaitForPendingFinalizers();
            }

            System.Diagnostics.Debugger.Break();
```

```
            Console.WriteLine($"Test completed, result={result}, entryPoint:
{testEntryPoint} unload success: {!testAlcWeakRef.IsAlive}");
        }
    }
}
```

Program loaded into the TestAssemblyLoadContext

The following code represents the *test.dll* passed to the `ExecuteAndUnload` method in the main testing program.

C#

```
using System;
using System.Runtime.InteropServices;
using System.Threading;

namespace test
{
    class Test
    {
    }

    class Program
    {
        public static void ThreadProc()
        {
            // Issue preventing unloading #4 - a thread running method
inside of the TestAssemblyLoadContext at the unload time
            Thread.Sleep(Timeout.Infinite);
        }

        static GCHandle handle;
        static int Main(string[] args)
        {
            // Issue preventing unloading #3 - normal GC handle
            handle = GCHandle.Alloc(new Test());
            Thread t = new Thread(new ThreadStart(ThreadProc));
            t.IsBackground = true;
            t.Start();
            Console.WriteLine($"Hello from the test: args[0] = {args[0]}");

            return 1;
        }
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How .NET is versioned

Article • 04/27/2023

The .NET Runtime and the .NET SDK add new features at different frequencies. In general, the SDK is updated more frequently than the Runtime. This article explains the runtime and the SDK version numbers.

.NET releases a new major version every November. Even-numbered releases, such as .NET 6 or .NET 8, are long-term supported (LTS). LTS releases get free support and patches for three years. Odd-numbered releases are standard-term support. Standard-term support releases get free support and patches for 18 months.

Versioning details

The .NET Runtime has a major.minor.patch approach to versioning that follows semantic versioning.

The .NET SDK, however, doesn't follow semantic versioning. The .NET SDK releases faster and its version numbers must communicate both the aligned runtime and the SDK's own minor and patch releases.

The first two positions of the .NET SDK version number match the .NET Runtime version it released with. Each version of the SDK can create applications for this runtime or any lower version.

The third position of the SDK version number communicates both the minor and patch number. The minor version is multiplied by 100. The final two digits represent the patch number. Minor version 1, patch version 2 would be represented as 102. For example, here's a possible sequence of runtime and SDK version numbers:

Change	.NET Runtime	.NET SDK (*)	Notes
Initial release	5.0.0	5.0.100	Initial release.
SDK patch	5.0.0	5.0.101	Runtime didn't change with this SDK patch. SDK patch bumps last digit in SDK patch.
Runtime and SDK patch	5.0.1	5.0.102	Runtime patch bumps Runtime patch number. SDK patch bumps last digit in SDK patch.
SDK feature change	5.0.1	5.0.200	Runtime patch didn't change. New SDK feature bumps first digit in SDK patch.

Change	.NET Runtime	.NET SDK (*)	Notes
Runtime patch	5.0.2	5.0.200	Runtime patch bumps Runtime patch number. SDK doesn't change.

From the preceding table you can see several policies:

- The Runtime and SDK share major and minor versions. The first two numbers for a given SDK and runtime should match. All the preceding examples are part of the .NET 5.0 release stream.
- The patch version of the runtime revs only when the runtime is updated. The SDK patch number doesn't update for a runtime patch.
- The patch version of the SDK updates only when the SDK is updated. It's possible that a runtime patch doesn't require an SDK patch.

NOTES:

- If the SDK has 10 feature updates before a runtime feature update, version numbers roll into the 1000 series. Version 5.0.1000 would follow version 5.0.900. This situation isn't expected to occur.
- 99 patch releases without a feature release won't occur. If a release approaches this number, it forces a feature release.

You can see more details in the initial proposal at the dotnet/designs repository.

Semantic versioning

The .NET *Runtime* roughly adheres to Semantic Versioning (SemVer), adopting the use of `MAJOR.MINOR.PATCH` versioning, using the various parts of the version number to describe the degree and type of change.

```
MAJOR.MINOR.PATCH[-PRERELEASE-BUILDNUMBER]
```

The optional `PRERELEASE` and `BUILDNUMBER` parts are never part of supported releases and only exist on nightly builds, local builds from source targets, and unsupported preview releases.

Understand runtime version number changes

- `MAJOR` is incremented once a year and may contain:
  - Significant changes in the product, or a new product direction.
  - API introduced breaking changes. There's a high bar to accepting breaking changes.
  - A newer `MAJOR` version of an existing dependency is adopted.

  Major releases happen once a year, even-numbered versions are long-term supported (LTS) releases. The first LTS release using this versioning scheme is .NET 6. The latest non-LTS version is .NET 5.

- `MINOR` is incremented when:
  - Public API surface area is added.
  - A new behavior is added.
  - A newer `MINOR` version of an existing dependency is adopted.
  - A new dependency is introduced.

- `PATCH` is incremented when:
  - Bug fixes are made.
  - Support for a newer platform is added.
  - A newer `PATCH` version of an existing dependency is adopted.
  - Any other change doesn't fit one of the previous cases.

When there are multiple changes, the highest element affected by individual changes is incremented, and the following ones are reset to zero. For example, when `MAJOR` is incremented, `MINOR.PATCH` are reset to zero. When `MINOR` is incremented, `PATCH` is reset to zero while `MAJOR` remains the same.

Version numbers in file names

The files downloaded for .NET carry the version, for example, `dotnet-sdk-5.0.301-win10-x64.exe`.

Preview versions

Preview versions have a `-preview.[number].[build]` appended to the version number. For example, `6.0.0-preview.5.21302.13`.

Servicing versions

After a release goes out, the release branches generally stop producing daily builds and instead start producing servicing builds. Servicing versions have a `-servicing-[number]` appended to the version. For example, `5.0.1-servicing-006924`.

.NET runtime compatibility

The .NET runtime maintains a high level of compatibility between versions. .NET apps should, by and large, continue to work after upgrading to a new major .NET runtime version.

Each major .NET runtime version contains intentional, carefully vetted, and documented breaking changes. The documented breaking changes aren't the only source of issues that can affect an app after upgrade. For example, a performance improvement in the .NET runtime (that's not considered a breaking change) can expose latent app threading bugs that cause the app to not work on that version. It's expected for large apps to require a few fixes after upgrading to a new .NET runtime major version.

By default, .NET apps are configured to run on a given .NET runtime major version, so recompilation is highly recommended to upgrade the app to run on a new .NET runtime major version. Then retest the app after upgrading to identify any issues.

Suppose upgrading via app recompilation isn't feasible. In that case, the .NET runtime provides additional settings to enable an app to run on a higher major .NET runtime version than the version it was compiled for. These settings don't change the risks involved in upgrading the app to a higher major .NET runtime version, and it's still required to retest the app post upgrade.

The .NET runtime supports loading libraries that target older .NET runtime versions. An app that's upgraded to a newer major .NET runtime version can reference libraries and NuGet packages that target older .NET runtime versions. It's unnecessary to simultaneously upgrade the target runtime version of all libraries and NuGet packages referenced by the app.

See also

- Breaking changes in .NET
- Target frameworks
- .NET distribution packaging
- .NET Support Lifecycle Fact Sheet
- Docker images for .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Select the .NET version to use

Article • 02/28/2023

This article explains the policies used by the .NET tools, SDK, and runtime for selecting versions. These policies provide a balance between running applications using the specified versions and enabling ease of upgrading both developer and end-user machines. These policies enable:

- Easy and efficient deployment of .NET, including security and reliability updates.
- Use the latest tools and commands independent of target runtime.

Version selection occurs:

- When you run an SDK command, the SDK uses the latest installed version.
- When you build an assembly, target framework monikers define build time APIs.
- When you run a .NET application, target framework dependent apps roll-forward.
- When you publish a self-contained application, self-contained deployments include the selected runtime.

The rest of this document examines those four scenarios.

The SDK uses the latest installed version

SDK commands include `dotnet new` and `dotnet run`. The .NET CLI must choose an SDK version for every `dotnet` command. It uses the latest SDK installed on the machine by default, even if:

- The project targets an earlier version of the .NET runtime.
- The latest version of the .NET SDK is a preview version.

You can take advantage of the latest SDK features and improvements while targeting earlier .NET runtime versions. You can target different runtime versions of .NET using the same SDK tools.

On rare occasions, you may need to use an earlier version of the SDK. You specify that version in a *global.json* file. The "use latest" policy means you only use *global.json* to specify a .NET SDK version earlier than the latest installed version.

global.json can be placed anywhere in the file hierarchy. The CLI searches upward from the project directory for the first *global.json* it finds. You control which projects a given *global.json* applies to by its place in the file system. The .NET CLI searches for a *global.json* file iteratively navigating the path upward from the current working directory.

The first *global.json* file found specifies the version used. If that SDK version is installed, that version is used. If the SDK specified in the *global.json* isn't found, the .NET CLI uses matching rules to select a compatible SDK, or fails if none is found.

The following example shows the *global.json* syntax:

JSON

```
{
  "sdk": {
    "version": "5.0.0"
  }
}
```

The process for selecting an SDK version is:

1. `dotnet` searches for a *global.json* file iteratively reverse-navigating the path upward from the current working directory.
2. `dotnet` uses the SDK specified in the first *global.json* found.
3. `dotnet` uses the latest installed SDK if no *global.json* is found.

For more information about SDK version selection, see the Matching rules and rollForward sections of the global.json overview article.

Target Framework Monikers define build time APIs

You build your project against APIs defined in a **Target Framework Moniker** (TFM). You specify the target framework in the project file. Set the `TargetFramework` element in your project file as shown in the following example:

XML

```
<TargetFramework>net5.0</TargetFramework>
```

You may build your project against multiple TFMs. Setting multiple target frameworks is more common for libraries but can be done with applications as well. You specify a `TargetFrameworks` property (plural of `TargetFramework`). The target frameworks are semicolon-delimited as shown in the following example:

XML

```
<TargetFrameworks>net5.0;netcoreapp3.1;net47</TargetFrameworks>
```

A given SDK supports a fixed set of frameworks, capped to the target framework of the runtime it ships with. For example, the .NET 5 SDK includes the .NET 5 runtime, which is an implementation of the `net5.0` target framework. The .NET 5 SDK supports `netcoreapp2.0`, `netcoreapp2.1`, `netcoreapp3.0`, and so on, but not `net6.0` (or higher). You install the .NET 6 SDK to build for `net6.0`.

.NET Standard

.NET Standard was a way to target an API surface shared by different implementations of .NET. Starting with the release of .NET 5, which is an API standard itself, .NET Standard has little relevance, except for one scenario: .NET Standard is useful when you want to target both .NET and .NET Framework. .NET 5 implements all .NET Standard versions.

For more information, see .NET 5 and .NET Standard.

Framework-dependent apps roll-forward

When you run an application from source with dotnet run, from a **framework-dependent deployment** with dotnet myapp.dll, or from a **framework-dependent executable** with `myapp.exe`, the `dotnet` executable is the **host** for the application.

The host chooses the latest patch version installed on the machine. For example, if you specified `net5.0` in your project file, and `5.0.2` is the latest .NET runtime installed, the `5.0.2` runtime is used.

If no acceptable `5.0.*` version is found, a new `5.*` version is used. For example, if you specified `net5.0` and only `5.1.0` is installed, the application runs using the `5.1.0` runtime. This behavior is referred to as "minor version roll-forward." Lower versions also won't be considered. When no acceptable runtime is installed, the application won't run.

A few usage examples demonstrate the behavior, if you target 5.0:

- ✔ 5.0 is specified. 5.0.3 is the highest patch version installed. 5.0.3 is used.
- ✘ 5.0 is specified. No 5.0.* versions are installed. 3.1.1 is the highest runtime installed. An error message is displayed.
- ✔ 5.0 is specified. No 5.0.* versions are installed. 5.1.0 is the highest runtime version installed. 5.1.0 is used.
- ✘ 3.0 is specified. No 3.x versions are installed. 5.0.0 is the highest runtime installed. An error message is displayed.

Minor version roll-forward has one side-effect that may affect end users. Consider the following scenario:

1. The application specifies that 5.0 is required.
2. When run, version 5.0.* isn't installed, however, 5.1.0 is. Version 5.1.0 will be used.
3. Later, the user installs 5.0.3 and runs the application again, 5.0.3 will now be used.

It's possible that 5.0.3 and 5.1.0 behave differently, particularly for scenarios like serializing binary data.

Control roll-forward behavior

> Before overriding default roll-forward behavior, familiarize yourself with the level of .NET runtime compatibility.

The roll-forward behavior for an application can be configured in four different ways:

1. Project-level setting by setting the `<RollForward>` property:

   XML

   ```xml
   <PropertyGroup>
     <RollForward>LatestMinor</RollForward>
   </PropertyGroup>
   ```

2. The `*.runtimeconfig.json` file.

   This file is produced when you compile your application. If the `<RollForward>` property was set in the project, it's reproduced in the `*.runtimeconfig.json` file as the `rollForward` setting. Users can edit this file to change the behavior of your application.

   JSON

   ```json
   {
     "runtimeOptions": {
       "tfm": "net5.0",
       "rollForward": "LatestMinor",
       "framework": {
         "name": "Microsoft.NETCore.App",
         "version": "5.0.0"
       }
     }
   }
   ```

3. The `dotnet` command's `--roll-forward <value>` property.

   When you run an application, you can control the roll-forward behavior through the command line:

   .NET CLI

   ```
   dotnet run --roll-forward LatestMinor
   dotnet myapp.dll --roll-forward LatestMinor
   myapp.exe --roll-forward LatestMinor
   ```

4. The `DOTNET_ROLL_FORWARD` environment variable.

Precedence

Roll forward behavior is set by the following order when your app is run, higher numbered items taking precedence over lower numbered items:

1. First the `*.runtimeconfig.json` config file is evaluated.
2. Next, the `DOTNET_ROLL_FORWARD` environment variable is considered, overriding the previous check.
3. Finally, any `--roll-forward` parameter passed to the running application overrides everything else.

Values

However you set the roll-forward setting, use one of the following values to set the behavior:

Value	Description
`Minor`	**Default** if not specified. Roll-forward to the lowest higher minor version, if requested minor version is missing. If the requested minor version is present, then the `LatestPatch` policy is used.
`Major`	Roll-forward to the next available higher major version, and lowest minor version, if requested major version is missing. If the requested major version is present, then the `Minor` policy is used.
`LatestPatch`	Roll-forward to the highest patch version. This value disables minor version roll-forward.
`LatestMinor`	Roll-forward to highest minor version, even if requested minor version is present.

Value	Description
`LatestMajor`	Roll-forward to highest major and highest minor version, even if requested major is present.
`Disable`	Don't roll-forward, only bind to the specified version. This policy isn't recommended for general use since it disables the ability to roll-forward to the latest patches. This value is only recommended for testing.

Self-contained deployments include the selected runtime

You can publish an application as a **self-contained distribution**. This approach bundles the .NET runtime and libraries with your application. Self-contained deployments don't have a dependency on runtime environments. Runtime version selection occurs at publishing time, not run time.

The *restore* event that occurs when publishing selects the latest patch version of the given runtime family. For example, `dotnet publish` will select .NET 5.0.3 if it's the latest patch version in the .NET 5 runtime family. The target framework (including the latest installed security patches) is packaged with the application.

An error occurs if the minimum version specified for an application isn't satisfied. `dotnet publish` binds to the latest runtime patch version (within a given major.minor version family). `dotnet publish` doesn't support the roll-forward semantics of `dotnet run`. For more information about patches and self-contained deployments, see the article on runtime patch selection in deploying .NET applications.

Self-contained deployments may require a specific patch version. You can override the minimum runtime patch version (to higher or lower versions) in the project file, as shown in the following example:

XML

```xml
<PropertyGroup>
  <RuntimeFrameworkVersion>5.0.7</RuntimeFrameworkVersion>
</PropertyGroup>
```

The `RuntimeFrameworkVersion` element overrides the default version policy. For self-contained deployments, the `RuntimeFrameworkVersion` specifies the *exact* runtime framework version. For framework-dependent applications, the `RuntimeFrameworkVersion` specifies the *minimum* required runtime framework version.

See also

- Download and install .NET.
- How to remove the .NET Runtime and SDK.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

.NET Runtime configuration settings

Article • 11/11/2023

.NET 5+ (including .NET Core versions) supports the use of configuration files and environment variables to configure the behavior of .NET applications.

> **Note**
>
> The articles in this section concern configuration of the .NET Runtime itself. If you're migrating to .NET Core 3.1 or later and are looking for a replacement for the *app.config* file, or if you simply want a way to use custom configuration values in your .NET app, see the **Microsoft.Extensions.Configuration.ConfigurationBuilder** class and **Configuration in .NET**.

Using these settings is an attractive option if:

- You don't own or control the source code for an application and therefore are unable to configure it programmatically.
- Multiple instances of your application run at the same time on a single system, and you want to configure each for optimum performance.

.NET provides the following mechanisms for configuring behavior of the .NET runtime:

- The runtimeconfig.json file
- MSBuild properties
- Environment variables

> **Tip**
>
> Configuring an option by using an environment variable applies the setting to all .NET apps. Configuring an option in the *runtimeconfig.json* or project file applies the setting to that application only.

Some configuration values can also be set programmatically by calling the AppContext.SetSwitch method.

The articles in this section of the documentation are organized by category, for example, debugging and garbage collection. Where applicable, configuration options are shown for *runtimeconfig.json* files, MSBuild properties, environment variables, and, for cross-reference, *app.config* files for .NET Framework projects.

runtimeconfig.json

When a project is built, an *[appname].runtimeconfig.json* file is generated in the output directory. If a *runtimeconfig.template.json* file exists in the same folder as the project file, any configuration options it contains are inserted into the *[appname].runtimeconfig.json* file. If you're building the app yourself, put any configuration options in the *runtimeconfig.template.json* file. If you're just running the app, insert them directly into the *[appname].runtimeconfig.json* file.

> **Note**
>
> - The *[appname].runtimeconfig.json* file will get overwritten on subsequent builds.
> - If your app's `OutputType` is not `Exe` and you want configuration options to be copied from *runtimeconfig.template.json* to *[appname].runtimeconfig.json*, you must explicitly set `GenerateRuntimeConfigurationFiles` to `true` in your project file. For apps that require a *runtimeconfig.json* file, this property defaults to `true`.

Specify runtime configuration options in the **configProperties** section of the *runtimeconfig.json* or *runtimeconfig.template.json* file. This section has the form:

JSON

```
"configProperties": {
  "config-property-name1": "config-value1",
  "config-property-name2": "config-value2"
}
```

Example [appname].runtimeconfig.json file

If you're placing the options in the output JSON file, nest them under the `runtimeOptions` property.

JSON

```
{
  "runtimeOptions": {
    "tfm": "netcoreapp3.1",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "3.1.0"
```

```
    },
    "configProperties": {
      "System.Globalization.UseNls": true,
      "System.Net.DisableIPv6": true,
      "System.GC.Concurrent": false,
      "System.Threading.ThreadPool.MinThreads": 4,
      "System.Threading.ThreadPool.MaxThreads": 25
    }
  }
}
```

Example runtimeconfig.template.json file

If you're placing the options in the template JSON file, omit the `runtimeOptions` property.

JSON

```
{
  "configProperties": {
    "System.Globalization.UseNls": true,
    "System.Net.DisableIPv6": true,
    "System.GC.Concurrent": false,
    "System.Threading.ThreadPool.MinThreads": "4",
    "System.Threading.ThreadPool.MaxThreads": "25"
  }
}
```

MSBuild properties

Some runtime configuration options can be set using MSBuild properties in the *.csproj* or *.vbproj* file of SDK-style .NET projects. MSBuild properties take precedence over options set in the *runtimeconfig.template.json* file.

For runtime configuration settings that don't have a specific MSBuild property, you can use the `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute.

Here is an example SDK-style project file with MSBuild properties for configuring the behavior of the .NET runtime:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
```

```
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <PropertyGroup>
    <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
    <ThreadPoolMinThreads>4</ThreadPoolMinThreads>
    <ThreadPoolMaxThreads>25</ThreadPoolMaxThreads>
  </PropertyGroup>

  <ItemGroup>
    <RuntimeHostConfigurationOption Include="System.Globalization.UseNls"
Value="true" />
    <RuntimeHostConfigurationOption Include="System.Net.DisableIPv6"
Value="true" />
  </ItemGroup>

</Project>
```

MSBuild properties for configuring the behavior of the runtime are noted in the individual articles for each area, for example, garbage collection. They're also listed in the Runtime configuration section of the MSBuild properties reference for SDK-style projects.

Environment variables

Environment variables can be used to supply some runtime configuration information. Configuration knobs specified as environment variables generally have the prefix **DOTNET_**.

> ⓘ **Note**
>
> .NET 6 standardizes on the prefix `DOTNET_` instead of `COMPlus_` for environment variables that configure .NET run-time behavior. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix for environment variables.

You can define environment variables from the Windows Control Panel, at the command line, or programmatically by calling the Environment.SetEnvironmentVariable(String, String) method on both Windows and Unix-based systems.

The following examples show how to set an environment variable at the command line:

```
shell
```

```
# Windows
set DOTNET_GCRetainVM=1

# Powershell
$env:DOTNET_GCRetainVM="1"

# Unix
export DOTNET_GCRetainVM=1
```

See also

- .NET environment variables

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Runtime configuration options for compilation

Article • 11/11/2023

This article details the settings you can use to configure .NET compilation.

> **Note**
>
> .NET 6 standardizes on the prefix `DOTNET_` instead of `COMPlus_` for environment variables that configure .NET run-time behavior. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix for environment variables.

Tiered compilation

- Configures whether the just-in-time (JIT) compiler uses tiered compilation. Tiered compilation transitions methods through two tiers:
  - The first tier generates code more quickly (quick JIT) or loads pre-compiled code (ReadyToRun).
  - The second tier generates optimized code in the background ("optimizing JIT").
- In .NET Core 3.0 and later, tiered compilation is enabled by default.
- In .NET Core 2.1 and 2.2, tiered compilation is disabled by default.
- For more information, see the Tiered compilation guide.

	Setting name	Values
runtimeconfig.json	`System.Runtime.TieredCompilation`	`true` - enabled `false` - disabled
MSBuild property	`TieredCompilation`	`true` - enabled `false` - disabled
Environment variable	`COMPlus_TieredCompilation` or `DOTNET_TieredCompilation`	`1` - enabled `0` - disabled

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Runtime.TieredCompilation": false
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.Runtime.TieredCompilation": false
   }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TieredCompilation>false</TieredCompilation>
  </PropertyGroup>

</Project>
```

Quick JIT

- Configures whether the JIT compiler uses *quick JIT*. For methods that don't contain loops and for which pre-compiled code is not available, quick JIT compiles them more quickly but without optimizations.
- Enabling quick JIT decreases startup time but can produce code with degraded performance characteristics. For example, the code may use more stack space, allocate more memory, and run slower.
- If quick JIT is disabled but tiered compilation is enabled, only pre-compiled code participates in tiered compilation. If a method is not pre-compiled with ReadyToRun, the JIT behavior is the same as if tiered compilation were disabled.
- In .NET Core 3.0 and later, quick JIT is enabled by default.
- In .NET Core 2.1 and 2.2, quick JIT is disabled by default.

	Setting name	Values
runtimeconfig.json	`System.Runtime.TieredCompilation.QuickJit`	`true` - enabled `false` - disabled
MSBuild property	`TieredCompilationQuickJit`	`true` - enabled `false` - disabled
Environment variable	`COMPlus_TC_QuickJit` or `DOTNET_TC_QuickJit`	`1` - enabled `0` - disabled

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Runtime.TieredCompilation.QuickJit": false
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.Runtime.TieredCompilation.QuickJit": false
   }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TieredCompilationQuickJit>false</TieredCompilationQuickJit>
  </PropertyGroup>

</Project>
```

Quick JIT for loops

- Configures whether the JIT compiler uses quick JIT on methods that contain loops.
- Enabling quick JIT for loops may improve startup performance. However, long-running loops can get stuck in less-optimized code for long periods.
- If quick JIT is disabled, this setting has no effect.
- If you omit this setting, quick JIT is not used for methods that contain loops. This is equivalent to setting the value to `false`.

	Setting name	Values
runtimeconfig.json	`System.Runtime.TieredCompilation.QuickJitForLoops`	`false` - disabled `true` - enabled
MSBuild property	`TieredCompilationQuickJitForLoops`	`false` - disabled `true` - enabled
Environment variable	`COMPlus_TC_QuickJitForLoops` or `DOTNET_TC_QuickJitForLoops`	`0` - disabled `1` - enabled

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Runtime.TieredCompilation.QuickJitForLoops": false
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.Runtime.TieredCompilation.QuickJitForLoops": false
```

```
    }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>

<TieredCompilationQuickJitForLoops>true</TieredCompilationQuickJitForLoops>
  </PropertyGroup>

</Project>
```

ReadyToRun

- Configures whether the .NET Core runtime uses pre-compiled code for images with available ReadyToRun data. Disabling this option forces the runtime to JIT-compile framework code.
- For more information, see Ready to Run.
- If you omit this setting, .NET uses ReadyToRun data when it's available. This is equivalent to setting the value to `1`.

	Setting name	Values
Environment variable	`COMPlus_ReadyToRun` or `DOTNET_ReadyToRun`	`1` - enabled `0` - disabled

Profile-guided optimization

This setting enables dynamic or tiered profile-guided optimization (PGO) in .NET 6 and later versions.

	Setting name	Values
Environment variable	`DOTNET_TieredPGO`	`1` - enabled `0` - disabled
MSBuild property	`TieredPGO`	`true` - enabled `false` - disabled

Examples

Project file:

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TieredPGO>true</TieredPGO>
  </PropertyGroup>

</Project>
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Runtime configuration options for debugging and profiling

Article • 12/05/2023

This article details the settings you can use to configure .NET debugging and profiling.

> **Note**
>
> .NET 6 standardizes on the prefix `DOTNET_` instead of `COMPlus_` for environment variables that configure .NET run-time behavior. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix for environment variables.

Enable diagnostics

- Configures whether the debugger, the profiler, and EventPipe diagnostics are enabled or disabled.
- If you omit this setting, diagnostics are enabled. This is equivalent to setting the value to `1`.

Expand table

	Setting name	Values
runtimeconfig.json	N/A	N/A
Environment variable	`COMPlus_EnableDiagnostics` or `DOTNET_EnableDiagnostics`	`1` - enabled `0` - disabled

Enable profiling

- Configures whether profiling is enabled for the currently running process.
- If you omit this setting, profiling is disabled. This is equivalent to setting the value to `0`.

Expand table

	Setting name	Values
runtimeconfig.json	N/A	N/A
Environment variable	`CORECLR_ENABLE_PROFILING`	`0` - disabled `1` - enabled

Profiler GUID

- Specifies the GUID of the profiler to load into the currently running process.

Expand table

	Setting name	Values
runtimeconfig.json	N/A	N/A
Environment variable	`CORECLR_PROFILER`	*string-guid*

Profiler location

- Specifies the path to the profiler DLL to load into the currently running process (or 32-bit or 64-bit process).
- If more than one variable is set, the bitness-specific variables take precedence. They specify which bitness of profiler to load.
- For more information, see Finding the profiler library.

Expand table

	Setting name	Values
Environment variable	`CORECLR_PROFILER_PATH`	*string-path*
Environment variable	`CORECLR_PROFILER_PATH_32`	*string-path*
Environment variable	`CORECLR_PROFILER_PATH_64`	*string-path*

Export perf maps and jit dumps

- Enables or disables selective enablement of perf maps or jit dumps. These files allow third party tools, such as the Linux `perf` tool, to identify call sites for dynamically generated code and precompiled ReadyToRun (R2R) modules.

- If you omit this setting, writing perf map and jit dump files are both disabled. This is equivalent to setting the value to `0`.
- When perf maps are disabled, not all managed callsites will be properly resolved.
- Depending on the Linux kernel version, both formats are supported by the `perf` tool.
- Enabling perf maps or jit dumps causes a 10-20% overhead. To minimize performance impact, it's recommended to selectively enable either perf maps or jit dumps, but not both.

The following table compares perf maps and jit maps.

Expand table

Format	Description	Supported on
Perf maps	Emits `/tmp/perf-<pid>.map`, which contains symbolic information for dynamically generated code. Emits `/tmp/perfinfo-<pid>.map`, which includes ReadyToRun (R2R) module symbol information and is used by PerfCollect.	Perf maps are supported on all Linux kernel versions.
Jit dumps	The jit dump format supersedes perf maps and contains more detailed symbolic information. When enabled, jit dumps are output to `/tmp/jit-<pid>.dump` files.	Linux kernel versions 5.4 or higher.

Expand table

	Setting name	Values
runtimeconfig.json	N/A	N/A
Environment variable	`COMPlus_PerfMapEnabled` or `DOTNET_PerfMapEnabled`	`0` - disabled `1` - perf maps and jit dumps both enabled `2` - jit dumps enabled `3` - perf maps enabled

Perf log markers

- Enables or disables the specified signal to be accepted and ignored as a marker in the perf logs.
- If you omit this setting, the specified signal is not ignored. This is equivalent to setting the value to `0`.

Expand table

	Setting name	Values
runtimeconfig.json	N/A	N/A
Environment variable	`COMPlus_PerfMapIgnoreSignal` or `DOTNET_PerfMapIgnoreSignal`	`0` - disabled `1` - enabled

> **Note**
>
> This setting is ignored if **DOTNET_PerfMapEnabled** is omitted or set to `0` (that is, disabled).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Runtime configuration options for garbage collection

Article • 11/11/2023

This page contains information about settings for the .NET runtime garbage collector (GC). If you're trying to achieve peak performance of a running app, consider using these settings. However, the defaults provide optimum performance for most applications in typical situations.

Settings are arranged into groups on this page. The settings within each group are commonly used in conjunction with each other to achieve a specific result.

> **Note**
>
> - These configurations are only read by the runtime when the GC is initialized (usually this means during the process startup time). If you change an environment variable when a process is already running, the change won't be reflected in that process. Settings that can be changed through APIs at run time, such as **latency level**, are omitted from this page.
> - Because GC is per process, it rarely ever makes sense to set these configurations at the machine level. For example, you wouldn't want every .NET process on a machine to use server GC or the same heap hard limit.
> - For number values, use decimal notation for settings in the *runtimeconfig.json* or *runtimeconfig.template.json* file and hexadecimal notation for environment variable settings. For hexadecimal values, you can specify them with or without the "0x" prefix.
> - If you're using the environment variables, .NET 6 and later versions standardize on the prefix `DOTNET_` instead of `COMPlus_`. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix, for example, `COMPlus_gcServer`.

Ways to specify the configuration

For different versions of the .NET runtime, there are different ways to specify the configuration values. The following table shows a summary.

Expand table

Config location	.NET versions this location applies to	Formats	How it's interpreted
runtimeconfig.json file/ runtimeconfig.template.json file	.NET (Core)	n	n is interpreted as a decimal value.
Environment variable	.NET Framework, .NET (Core)	0xn or n	n is interpreted as a hex value in either format
app.config file	.NET Framework	0xn	n is interpreted as a hex value[1]

[1] You can specify a value without the `0x` prefix for an app.config file setting, but it's not recommended. On .NET Framework 4.8+, due to a bug, a value specified without the `0x` prefix is interpreted as hexadecimal, but on previous versions of .NET Framework, it's interpreted as decimal. To avoid having to change your config, use the `0x` prefix when specifying a value in your app.config file.

For example, to specify 12 heaps for `GCHeapCount` for a .NET Framework app named *A.exe*, add the following XML to the *A.exe.config* file.

XML

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    ...
    <runtime>
        <gcServer enabled="true"/>
        <GCHeapCount>0xc</GCHeapCount>
    </runtime>
</configuration>
```

For both .NET (Core) and .NET Framework, you can use environment variables.

On Windows using .NET 6 or a later version:

Windows Command Prompt

```
SET DOTNET_gcServer=1
SET DOTNET_GCHeapCount=c
```

On Windows using .NET 5 or earlier:

Windows Command Prompt

```
SET COMPlus_gcServer=1
SET COMPlus_GCHeapCount=c
```

On other operating systems:

For .NET 6 or later versions:

Bash

```
export DOTNET_gcServer=1
export DOTNET_GCHeapCount=c
```

For .NET 5 and earlier versions:

Bash

```
export COMPlus_gcServer=1
export COMPlus_GCHeapCount=c
```

If you're not using .NET Framework, you can also set the value in the *runtimeconfig.json* or *runtimeconfig.template.json* file.

runtimeconfig.json file:

JSON

```
{
  "runtimeOptions": {
   "configProperties": {
      "System.GC.Server": true,
      "System.GC.HeapCount": 12
   }
  }
}
```

runtimeconfig.template.json file:

JSON

```
{
  "configProperties": {
    "System.GC.Server": true,
    "System.GC.HeapCount": 12
  }
}
```

Flavors of garbage collection

The two main flavors of garbage collection are workstation GC and server GC. For more information about differences between the two, see Workstation and server garbage collection.

The subflavors of garbage collection are background and non-concurrent.

Use the following settings to select flavors of garbage collection:

- Workstation vs. server GC
- Background GC

Workstation vs. server

- Configures whether the application uses workstation garbage collection or server garbage collection.
- Default: Workstation garbage collection. This is equivalent to setting the value to `false`.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.Server`	`false` - workstation `true` - server	.NET Core 1.0
MSBuild property	`ServerGarbageCollection`	`false` - workstation `true` - server	.NET Core 1.0
Environment variable	`COMPlus_gcServer`	`0` - workstation `1` - server	.NET Core 1.0
Environment variable	`DOTNET_gcServer`	`0` - workstation `1` - server	.NET 6
app.config for .NET Framework	GCServer	`false` - workstation `true` - server	

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.Server": true
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.GC.Server": true
   }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ServerGarbageCollection>true</ServerGarbageCollection>
  </PropertyGroup>

</Project>
```

Background GC

- Configures whether background (concurrent) garbage collection is enabled.
- Default: Use background GC. This is equivalent to setting the value to `true`.
- For more information, see Background garbage collection.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.Concurrent`	`true` - background GC `false` - non-concurrent GC	.NET Core 1.0
MSBuild property	`ConcurrentGarbageCollection`	`true` - background GC `false` - non-concurrent GC	.NET Core 1.0
Environment variable	`COMPlus_gcConcurrent`	`1` - background GC `0` - non-concurrent GC	.NET Core 1.0
Environment variable	`DOTNET_gcConcurrent`	`1` - background GC `0` - non-concurrent GC	.NET 6
app.config for .NET Framework	gcConcurrent	`true` - background GC `false` - non-concurrent GC	

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.Concurrent": false
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
```

```
      "System.GC.Concurrent": false
    }
}
```

Project file:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
  </PropertyGroup>

</Project>
```

Manage resource usage

Use the following settings to manage the garbage collector's memory and processor usage:

- Affinitize
- Affinitize mask
- Affinitize ranges
- CPU groups
- Heap count
- Heap limit
- Heap limit percent
- High memory percent
- Per-object-heap limits
- Per-object-heap limit percents
- Retain VM

For more information about some of these settings, see the Middle ground between workstation and server GC blog entry.

Heap count

- Limits the number of heaps created by the garbage collector.
- Applies to server garbage collection only.
- If GC processor affinity is enabled, which is the default, the heap count setting affinitizes `n` GC heaps/threads to the first `n` processors. (Use the affinitize mask or affinitize ranges settings to specify exactly which processors to affinitize.)

- If GC processor affinity is disabled, this setting limits the number of GC heaps.
- For more information, see the GCHeapCount remarks.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HeapCount`	*decimal value*	.NET Core 3.0
Environment variable	`COMPlus_GCHeapCount`	*hexadecimal value*	.NET Core 3.0
Environment variable	`DOTNET_GCHeapCount`	*hexadecimal value*	.NET 6
app.config for .NET Framework	GCHeapCount	*decimal value*	.NET Framework 4.6.2

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapCount": 16
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.GC.HeapCount": 16
   }
}
```

> **Tip**
>
> If you're setting the option in *runtimeconfig.json*, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For example, to limit the number of heaps to 16, the values would be 16 for the JSON file and 0x10 or 10 for the environment variable.

Affinitize mask

- Specifies the exact processors that garbage collector threads should use.
- If GC processor affinity is disabled, this setting is ignored.
- Applies to server garbage collection only.
- The value is a bit mask that defines the processors that are available to the process. For example, a decimal value of 1023 (or a hexadecimal value of 0x3FF or 3FF if you're using the environment variable) is 0011 1111 1111 in binary notation. This specifies that the first 10 processors are to be used. To specify the next 10 processors, that is, processors 10-19, specify a decimal value of 1047552 (or a hexadecimal value of 0xFFC00 or FFC00), which is equivalent to a binary value of 1111 1111 1100 0000 0000.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HeapAffinitizeMask`	*decimal value*	.NET Core 3.0
Environment variable	`COMPlus_GCHeapAffinitizeMask`	*hexadecimal value*	.NET Core 3.0
Environment variable	`DOTNET_GCHeapAffinitizeMask`	*hexadecimal value*	.NET 6
app.config for .NET Framework	GCHeapAffinitizeMask	*decimal value*	.NET Framework 4.6.2

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapAffinitizeMask": 1023
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.GC.HeapAffinitizeMask": 1023
   }
}
```

Affinitize ranges

- Specifies the list of processors to use for garbage collector threads.
- This setting is similar to System.GC.HeapAffinitizeMask, except it allows you to specify more than 64 processors.
- For Windows operating systems, prefix the processor number or range with the corresponding CPU group, for example, "0:1-10,0:12,1:50-52,1:7". If you don't actually have more than 1 CPU group, you can't use this setting. You must use the Affinitize mask setting. And the numbers you specify are within that group, which means it cannot be >= 64.
- For Linux operating systems, where the CPU group concept doesn't exist, you can use both this setting and the Affinitize mask setting to specify the same ranges. And instead of "0:1-10", specify "1-10" because you don't need to specify a group index.
- If GC processor affinity is disabled, this setting is ignored.
- Applies to server garbage collection only.
- For more information, see Making CPU configuration better for GC on machines with > 64 CPUs on Maoni Stephens' blog.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HeapAffinitizeRanges`	Comma-separated list of processor numbers or ranges of processor numbers. Unix example: "1-10,12,50-52,70" Windows example: "0:1-10,0:12,1:50-52,1:7"	.NET Core 3.0
Environment variable	`COMPlus_GCHeapAffinitizeRanges`	Comma-separated list of processor numbers or ranges of processor numbers. Unix example: "1-10,12,50-52,70" Windows example: "0:1-10,0:12,1:50-52,1:7"	.NET Core 3.0
Environment variable	`DOTNET_GCHeapAffinitizeRanges`	Comma-separated list of processor numbers or ranges of processor numbers. Unix example: "1-10,12,50-52,70" Windows example: "0:1-10,0:12,1:50-52,1:7"	.NET 6

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapAffinitizeRanges": "0:1-10,0:12,1:50-52,1:7"
```

```
        }
    }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.GC.HeapAffinitizeRanges": "0:1-10,0:12,1:50-52,1:7"
   }
}
```

CPU groups

- Configures whether the garbage collector uses CPU groups or not.

  When a 64-bit Windows computer has multiple CPU groups, that is, there are more than 64 processors, enabling this element extends garbage collection across all CPU groups. The garbage collector uses all cores to create and balance heaps.

  > **Note**
  >
  > This is a Windows-only concept. In older Windows versions, Windows limited a process to one CPU group. Thus, GC only used one CPU group unless you used this setting to enable multiple CPU groups. This OS limitation was lifted in Windows 11 and Server 2022. Also, starting in .NET 7, GC by default uses all CPU groups when running on Windows 11 or Server 2022.

- Applies to server garbage collection on 64-bit Windows operating systems only.
- Default: GC does not extend across CPU groups. This is equivalent to setting the value to `0`.
- For more information, see Making CPU configuration better for GC on machines with > 64 CPUs on Maoni Stephens' blog.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.CpuGroup`	`false` - disabled	.NET 5

	Setting name	Values	Version introduced
		`true` - enabled	
Environment variable	`COMPlus_GCCpuGroup`	`0` - disabled `1` - enabled	.NET Core 1.0
Environment variable	`DOTNET_GCCpuGroup`	`0` - disabled `1` - enabled	.NET 6
app.config for .NET Framework	GCCpuGroup	`false` - disabled `true` - enabled	

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

> **Note**
>
> To configure the common language runtime (CLR) to also distribute threads from the thread pool across all CPU groups, enable the **Thread_UseAllCpuGroups element** option. For .NET Core apps, you can enable this option by setting the value of the `DOTNET_Thread_UseAllCpuGroups` environment variable to `1`.

Affinitize

- Specifies whether to *affinitize* garbage collection threads with processors. To affinitize a GC thread means that it can only run on its specific CPU. A heap is created for each GC thread.
- Applies to server garbage collection only.
- Default: Affinitize garbage collection threads with processors. This is equivalent to setting the value to `false`.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.NoAffinitize`	`false` - affinitize `true` - don't affinitize	.NET Core 3.0

	Setting name	Values	Version introduced
Environment variable	`COMPlus_GCNoAffinitize`	`0` - affinitize `1` - don't affinitize	.NET Core 3.0
Environment variable	`DOTNET_GCNoAffinitize`	`0` - affinitize `1` - don't affinitize	.NET 6
app.config for .NET Framework	GCNoAffinitize	`false` - affinitize `true` - don't affinitize	.NET Framework 4.6.2

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.NoAffinitize": true
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.GC.NoAffinitize": true
   }
}
```

Heap limit

- Specifies the maximum commit size, in bytes, for the GC heap and GC bookkeeping.
- This setting only applies to 64-bit computers.
- This setting is ignored if the Per-object-heap limits are configured.
- The default value, which only applies in certain cases, is the greater of 20 MB or 75% of the memory limit on the container. The default value applies if:
  - The process is running inside a container that has a specified memory limit.
  - System.GC.HeapHardLimitPercent is not set.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HeapHardLimit`	*decimal value*	.NET Core 3.0
Environment variable	`COMPlus_GCHeapHardLimit`	*hexadecimal value*	.NET Core 3.0
Environment variable	`DOTNET_GCHeapHardLimit`	*hexadecimal value*	.NET 6

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapHardLimit": 209715200
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.GC.HeapHardLimit": 209715200
   }
}
```

> **Tip**
>
> If you're setting the option in *runtimeconfig.json*, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For example, to specify a heap hard limit of 200 mebibytes (MiB), the values would be 209715200 for the JSON file and 0xC800000 or C800000 for the environment variable.

Heap limit percent

- Specifies the allowable GC heap usage as a percentage of the total physical memory.
- If System.GC.HeapHardLimit is also set, this setting is ignored.
- This setting only applies to 64-bit computers.
- If the process is running inside a container that has a specified memory limit, the percentage is calculated as a percentage of that memory limit.
- This setting is ignored if the Per-object-heap limits are configured.
- The default value, which only applies in certain cases, is the greater of 20 MB or 75% of the memory limit on the container. The default value applies if:
  - The process is running inside a container that has a specified memory limit.
  - System.GC.HeapHardLimit is not set.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HeapHardLimitPercent`	*decimal value*	.NET Core 3.0
Environment variable	`COMPlus_GCHeapHardLimitPercent`	*hexadecimal value*	.NET Core 3.0

	Setting name	Values	Version introduced
Environment variable	`DOTNET_GCHeapHardLimitPercent`	*hexadecimal value*	.NET 6

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapHardLimitPercent": 30
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.GC.HeapHardLimitPercent": 30
   }
}
```

Tip

If you're setting the option in *runtimeconfig.json*, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For example, to limit the heap usage to 30%, the values would be 30 for the JSON file and 0x1E or 1E for the environment variable.

Per-object-heap limits

You can specify the GC's allowable heap usage on a per-object-heap basis. The different heaps are the large object heap (LOH), small object heap (SOH), and pinned object heap (POH).

- If you specify a value for any of the `DOTNET_GCHeapHardLimitSOH`, `DOTNET_GCHeapHardLimitLOH`, or `DOTNET_GCHeapHardLimitPOH` settings, you must also specify a value for `DOTNET_GCHeapHardLimitSOH` and `DOTNET_GCHeapHardLimitLOH`. If you don't, the runtime will fail to initialize.
- The default value for `DOTNET_GCHeapHardLimitPOH` is 0. `DOTNET_GCHeapHardLimitSOH` and `DOTNET_GCHeapHardLimitLOH` don't have default values.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HeapHardLimitSOH`	*decimal value*	.NET 5
Environment variable	`COMPlus_GCHeapHardLimitSOH`	*hexadecimal value*	.NET 5
Environment variable	`DOTNET_GCHeapHardLimitSOH`	*hexadecimal value*	.NET 6

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HeapHardLimitLOH`	*decimal value*	.NET 5
Environment variable	`COMPlus_GCHeapHardLimitLOH`	*hexadecimal value*	.NET 5
Environment variable	`DOTNET_GCHeapHardLimitLOH`	*hexadecimal value*	.NET 6

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HeapHardLimitPOH`	*decimal value*	.NET 5
Environment variable	`COMPlus_GCHeapHardLimitPOH`	*hexadecimal value*	.NET 5
Environment variable	`DOTNET_GCHeapHardLimitPOH`	*hexadecimal value*	.NET 6

These configuration settings don't have specific MSBuild properties. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

> **Tip**
>
> If you're setting the option in *runtimeconfig.json*, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For example, to specify a heap hard limit of 200 mebibytes (MiB), the values would be 209715200 for the JSON file and 0xC800000 or C800000 for the environment variable.

Per-object-heap limit percents

You can specify the GC's allowable heap usage on a per-object-heap basis. The different heaps are the large object heap (LOH), small object heap (SOH), and pinned object heap (POH).

- If you specify a value for any of the `DOTNET_GCHeapHardLimitSOHPercent`, `DOTNET_GCHeapHardLimitLOHPercent`, or `DOTNET_GCHeapHardLimitPOHPercent` settings, you must also specify a value for `DOTNET_GCHeapHardLimitSOHPercent` and `DOTNET_GCHeapHardLimitLOHPercent`. If you don't, the runtime will fail to initialize.
- These settings are ignored if `DOTNET_GCHeapHardLimitSOH`, `DOTNET_GCHeapHardLimitLOH`, and `DOTNET_GCHeapHardLimitPOH` are specified.
- A value of 1 means that GC uses 1% of total physical memory for that object heap.
- Each value must be greater than zero and less than 100. Additionally, the sum of the three percentage values must be less than 100. Otherwise, the runtime will fail to initialize.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HeapHardLimitSOHPercent`	*decimal value*	.NET 5
Environment variable	`COMPlus_GCHeapHardLimitSOHPercent`	*hexadecimal value*	.NET 5
Environment variable	`DOTNET_GCHeapHardLimitSOHPercent`	*hexadecimal value*	.NET 6

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HeapHardLimitLOHPercent`	*decimal value*	.NET 5
Environment variable	`COMPlus_GCHeapHardLimitLOHPercent`	*hexadecimal value*	.NET 5
Environment variable	`DOTNET_GCHeapHardLimitLOHPercent`	*hexadecimal value*	.NET 6

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HeapHardLimitPOHPercent`	*decimal value*	.NET 5
Environment variable	`COMPlus_GCHeapHardLimitPOHPercent`	*hexadecimal value*	.NET 5
Environment variable	`DOTNET_GCHeapHardLimitPOHPercent`	*hexadecimal value*	.NET 6

These configuration settings don't have specific MSBuild properties. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

> **Tip**
>
> If you're setting the option in *runtimeconfig.json*, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For example, to limit the heap usage to 30%, the values would be 30 for the JSON file and 0x1E or 1E for the environment variable.

High memory percent

Memory load is indicated by the percentage of physical memory in use. By default, when the physical memory load reaches **90%**, garbage collection becomes more aggressive about doing full, compacting garbage collections to avoid paging. When memory load is below 90%, GC favors background collections for full garbage collections, which have shorter pauses but don't reduce the total heap size by much. On

machines with a significant amount of memory (80GB or more), the default load threshold is between 90% and 97%.

The high memory load threshold can be adjusted by the `DOTNET_GCHighMemPercent` environment variable or `System.GC.HighMemoryPercent` JSON configuration setting. Consider adjusting the threshold if you want to control heap size. For example, for the dominant process on a machine with 64GB of memory, it's reasonable for GC to start reacting when there's 10% of memory available. But for smaller processes, for example, a process that only consumes 1GB of memory, GC can comfortably run with less than 10% of memory available. For these smaller processes, consider setting the threshold higher. On the other hand, if you want larger processes to have smaller heap sizes (even when there's plenty of physical memory available), lowering this threshold is an effective way for GC to react sooner to compact the heap down.

> **Note**
>
> For processes running in a container, GC considers the physical memory based on the container limit.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.HighMemoryPercent`	*decimal value*	.NET 5
Environment variable	`COMPlus_GCHighMemPercent`	*hexadecimal value*	.NET Core 3.0 .NET Framework 4.7.2
Environment variable	`DOTNET_GCHighMemPercent`	*hexadecimal value*	.NET 6

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

> **Tip**
>
> If you're setting the option in *runtimeconfig.json*, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For

example, to set the high memory threshold to 75%, the values would be 75 for the JSON file and 0x4B or 4B for the environment variable.

Retain VM

- Configures whether segments that should be deleted are put on a standby list for future use or are released back to the operating system (OS).
- Default: Release segments back to the operating system. This is equivalent to setting the value to `false`.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.RetainVM`	`false` - release to OS `true` - put on standby	.NET Core 1.0
MSBuild property	`RetainVMGarbageCollection`	`false` - release to OS `true` - put on standby	.NET Core 1.0
Environment variable	`COMPlus_GCRetainVM`	`0` - release to OS `1` - put on standby	.NET Core 1.0
Environment variable	`DOTNET_GCRetainVM`	`0` - release to OS `1` - put on standby	.NET 6

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.RetainVM": true
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.GC.RetainVM": true
   }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <RetainVMGarbageCollection>true</RetainVMGarbageCollection>
  </PropertyGroup>

</Project>
```

Large pages

- Specifies whether large pages should be used when a heap hard limit is set.
- Default: Don't use large pages when a heap hard limit is set. This is equivalent to setting the value to `0`.
- This is an experimental setting.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	N/A	N/A	N/A
Environment variable	`COMPlus_GCLargePages`	`0` - disabled `1` - enabled	.NET Core 3.0
Environment variable	`DOTNET_GCLargePages`	`0` - disabled `1` - enabled	.NET 6

Allow large objects

- Configures garbage collector support on 64-bit platforms for arrays that are greater than 2 gigabytes (GB) in total size.
- Default: GC supports arrays greater than 2-GB. This is equivalent to setting the value to `1`.
- This option may become obsolete in a future version of .NET.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	N/A	N/A	N/A
Environment variable	`COMPlus_gcAllowVeryLargeObjects`	`1` - enabled `0` - disabled	.NET Core 1.0
Environment variable	`DOTNET_gcAllowVeryLargeObjects`	`1` - enabled `0` - disabled	.NET 6
app.config for .NET Framework	gcAllowVeryLargeObjects	`1` - enabled `0` - disabled	.NET Framework 4.5

Large object heap threshold

- Specifies the threshold size, in bytes, that causes objects to go on the large object heap (LOH).
- The default threshold is 85,000 bytes.
- The value you specify must be larger than the default threshold.
- The value might be capped by the runtime to the maximum possible size for the current configuration. You can inspect the value in use at run time through the GC.GetConfigurationVariables() API.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.LOHThreshold`	*decimal value*	.NET Core 1.0

	Setting name	Values	Version introduced
Environment variable	`COMPlus_GCLOHThreshold`	*hexadecimal value*	.NET Core 1.0
Environment variable	`DOTNET_GCLOHThreshold`	*hexadecimal value*	.NET 6
app.config for .NET Framework	GCLOHThreshold	*decimal value*	.NET Framework 4.8

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.LOHThreshold": 120000
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.GC.LOHThreshold": 120000
   }
}
```

Tip

If you're setting the option in *runtimeconfig.json*, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For

example, to set a threshold size of 120,000 bytes, the values would be 120000 for the JSON file and 0x1D4C0 or 1D4C0 for the environment variable.

Standalone GC

- Specifies the name of a GC native library that the runtime loads in place of the default GC implementation. This native library needs to reside in the same directory as the .NET runtime (**coreclr.dll** on Windows, **libcoreclr.so** on Linux).

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	N/A	N/A	N/A
Environment variable	`COMPlus_GCName`	*string_path*	.NET Core 2.0
Environment variable	`DOTNET_GCName`	*string_path*	.NET 6

Conserve memory

- Configures the garbage collector to conserve memory at the expense of more frequent garbage collections and possibly longer pause times.
- Default value is 0 - this implies no change.
- Besides the default value 0, values between 1 and 9 (inclusive) are valid. The higher the value, the more the garbage collector tries to conserve memory and thus to keep the heap small.
- If the value is non-zero, the large object heap will be compacted automatically if it has too much fragmentation.

Expand table

	Setting name	Values	Version introduced
runtimeconfig.json	`System.GC.ConserveMemory`	`0` - `9`	.NET 6
Environment variable	`COMPlus_GCConserveMemory`	`0` - `9`	.NET Framework 4.8
Environment variable	`DOTNET_GCConserveMemory`	`0` - `9`	.NET 6
app.config for .NET Framework	GCConserveMemory	`0` - `9`	.NET Framework 4.8

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Example *app.config* file:

XML

```xml
<configuration>
  <runtime>
    <GCConserveMemory enabled="5"/>
  </runtime>
</configuration>
```

Tip

Experiment with different numbers to see which value works best for you. Start with a value between 5 and 7.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Runtime configuration options for globalization

Article • 11/11/2023

Invariant mode

- Determines whether a .NET Core app runs in globalization-invariant mode without access to culture-specific data and behavior.
- If you omit this setting, the app runs with access to cultural data. This is equivalent to setting the value to `false`.
- For more information, see .NET Core globalization invariant mode.

	Setting name	Values
runtimeconfig.json	`System.Globalization.Invariant`	`false` - access to cultural data `true` - run in invariant mode
MSBuild property	`InvariantGlobalization`	`false` - access to cultural data `true` - run in invariant mode
Environment variable	`DOTNET_SYSTEM_GLOBALIZATION_INVARIANT`	`0` - access to cultural data `1` - run in invariant mode

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Globalization.Invariant": true
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.Globalization.Invariant": true
```

```
    }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <InvariantGlobalization>true</InvariantGlobalization>
  </PropertyGroup>

</Project>
```

Era year ranges

- Determines whether range checks for calendars that support multiple eras are relaxed or whether dates that overflow an era's date range throw an ArgumentOutOfRangeException.
- If you omit this setting, range checks are relaxed. This is equivalent to setting the value to `false`.
- For more information, see Calendars, eras, and date ranges: Relaxed range checks.

	Setting name	Values
runtimeconfig.json	`Switch.System.Globalization.EnforceJapaneseEraYearRanges`	`false` - relaxed range checks `true` - overflows cause an exception
Environment variable	N/A	N/A

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Japanese date parsing

- Determines whether a string that contains either "1" or "Gannen" as the year parses successfully or whether only "1" is supported.

- If you omit this setting, strings that contain either "1" or "Gannen" as the year parse successfully. This is equivalent to setting the value to `false`.
- For more information, see Represent dates in calendars with multiple eras.

	Setting name	Values
runtimeconfig.json	`Switch.System.Globalization.EnforceLegacyJapaneseDateParsing`	`false` - "Gannen" or "1" is supported `true` - only "1" is supported
Environment variable	N/A	N/A

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Japanese year format

- Determines whether the first year of a Japanese calendar era is formatted as "Gannen" or as a number.
- If you omit this setting, the first year is formatted as "Gannen". This is equivalent to setting the value to `false`.
- For more information, see Represent dates in calendars with multiple eras.

	Setting name	Values
runtimeconfig.json	`Switch.System.Globalization.FormatJapaneseFirstYearAsANumber`	`false` - format as "Gannen" `true` - format as number
Environment variable	N/A	N/A

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

NLS

- Determines whether .NET uses National Language Support (NLS) or International Components for Unicode (ICU) globalization APIs for Windows apps. .NET 5 and later versions use ICU globalization APIs by default on Windows 10 May 2019 Update and later versions.
- If you omit this setting, .NET uses ICU globalization APIs by default. This is equivalent to setting the value to `false`.
- For more information, see Globalization APIs use ICU libraries on Windows.

	Setting name	Values	Introduced
runtimeconfig.json	`System.Globalization.UseNls`	`false` - Use ICU globalization APIs `true` - Use NLS globalization APIs	.NET 5
Environment variable	`DOTNET_SYSTEM_GLOBALIZATION_USENLS`	`false` - Use ICU globalization APIs `true` - Use NLS globalization APIs	.NET 5

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Predefined cultures

- Configures whether apps can create cultures other than the invariant culture when globalization-invariant mode is enabled.
- If you omit this setting, .NET restricts the creation of cultures in globalization-invariant mode. This is equivalent to setting the value to `true`.
- For more information, see Culture creation and case mapping in globalization-invariant mode.

	Setting name	Values	Introduced
runtimeconfig.json	`System.Globalization.PredefinedCulturesOnly`	`true` - In globalization-invariant mode, don't allow creation of any culture except the	.NET 6

	Setting name	Values	Introduced
		invariant culture. `false` - Allow creation of any culture.	
MSBuild property	`PredefinedCulturesOnly`	`true` - In globalization-invariant mode, don't allow creation of any culture except the invariant culture. `false` - Allow creation of any culture.	.NET 6
Environment variable	`DOTNET_SYSTEM_GLOBALIZATION_PREDEFINED_CULTURES_ONLY`	`true` - In globalization-invariant mode, don't allow creation of any culture except the invariant culture. `false` - Allow creation of any culture.	.NET 6

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Runtime configuration options for networking

Article • 11/11/2023

HTTP/2 protocol

- Configures whether support for the HTTP/2 protocol is enabled.
- Introduced in .NET Core 3.0.
- .NET Core 3.0 only: If you omit this setting, support for the HTTP/2 protocol is disabled. This is equivalent to setting the value to `false`.
- .NET Core 3.1 and .NET 5+: If you omit this setting, support for the HTTP/2 protocol is enabled. This is equivalent to setting the value to `true`.

	Setting name	Values
runtimeconfig.json	`System.Net.Http.SocketsHttpHandler.Http2Support`	`false` - disabled `true` - enabled
Environment variable	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_HTTP2SUPPORT`	`0` - disabled `1` - enabled

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

SPN creation in HttpClient (.NET 6 and later)

- Impacts generation of service principal names (SPN) for Kerberos and NTLM authentication when `Host` header is missing and target is not running on default port.
- .NET Core 2.x and 3.x do not include port in SPN.
- .NET Core 5.x does include port in SPN
- .NET 6 and later versions don't include the port, but the behavior is configurable.

	Setting name	Values
runtimeconfig.json	`System.Net.Http.UsePortInSpn`	`true` - includes port number in SPN, for example, `HTTP/host:port` `false` - does not include port in SPN, for example, `HTTP/host`
Environment variable	`DOTNET_SYSTEM_NET_HTTP_USEPORTINSPN`	`1` - includes port number in SPN, for example, `HTTP/host:port` `0` - does not include port in SPN, for example, `HTTP/host`

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

UseSocketsHttpHandler (.NET Core 2.1-3.1 only)

- Configures whether System.Net.Http.HttpClientHandler uses System.Net.Http.SocketsHttpHandler or older HTTP protocol stacks (WinHttpHandler on Windows and `CurlHandler`, an internal class implemented on top of libcurl, on Linux).

  > **Note**
  >
  > You may be using high-level networking APIs instead of directly instantiating the **HttpClientHandler** class. This setting also affects which HTTP protocol stack is used by high-level networking APIs, including **HttpClient** and **HttpClientFactory**.

- If you omit this setting, HttpClientHandler uses SocketsHttpHandler. This is equivalent to setting the value to `true`.

	Setting name	Values
runtimeconfig.json	`System.Net.Http.UseSocketsHttpHandler`	`true` - enables the use of SocketsHttpHandler `false` - enables the use of WinHttpHandler on Windows or libcurl on Linux

	Setting name	Values
Environment variable	`DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPHANDLER`	`1` - enables the use of SocketsHttpHandler `0` - enables the use of WinHttpHandler on Windows or libcurl on Linux

> **Note**
>
> Starting in .NET 5, the `System.Net.Http.UseSocketsHttpHandler` setting is no longer available.

Latin1 headers (.NET Core 3.1 only)

- This switch allows relaxing the HTTP header validation, enabling SocketsHttpHandler to send ISO-8859-1 (Latin-1) encoded characters in headers.
- If you omit this setting, an attempt to send a non-ASCII character will result in HttpRequestException. This is equivalent to setting the value to `false`.

	Setting name	Values
runtimeconfig.json	`System.Net.Http.SocketsHttpHandler.AllowLatin1Headers`	`false` - disabled `true` - enabled
Environment variable	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_ALLOWLATIN1HEADERS`	`0` - disabled `1` - enabled

> **Note**
>
> This option is only available in .NET Core 3.1 since version 3.1.9, and not in previous or later versions. In .NET 5 we recommend using **RequestHeaderEncodingSelector**.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Runtime configuration options for threading

Article • 09/23/2023

This article details the settings you can use to configure threading in .NET.

> **Note**
>
> .NET 6 standardizes on the prefix `DOTNET_` instead of `COMPlus_` for environment variables that configure .NET run-time behavior. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix for environment variables.

Use all CPU groups on Windows

- On machines that have multiple CPU groups, this setting configures whether components such as the thread pool use all CPU groups or only the primary CPU group of the process. The setting also affects what Environment.ProcessorCount returns.
- When this setting is enabled, all CPU groups are used and threads are also automatically distributed across CPU groups by default.
- This setting is enabled by default on Windows 11 and later versions, and disabled by default on Windows 10 and earlier versions. For this setting to take effect when enabled, the GC must also be configured to use all CPU groups; for more information, see GC CPU groups.

	Setting name	Values
runtimeconfig.json	N/A	N/A
Environment variable	`COMPlus_Thread_UseAllCpuGroups` or `DOTNET_Thread_UseAllCpuGroups`	`0` - disabled `1` - enabled

Assign threads to CPU groups on Windows

- On machines that have multiple CPU groups and all CPU groups are being used, this setting configures whether threads are automatically distributed across CPU groups.
- When this setting is enabled, new threads are assigned to a CPU group in a way that tries to fully populate a CPU group that is already in use before utilizing a new CPU group.
- This setting is enabled by default.

	Setting name	Values
runtimeconfig.json	N/A	N/A
Environment variable	`COMPlus_Thread_AssignCpuGroups` or `DOTNET_Thread_AssignCpuGroups`	`0` - disabled `1` - enabled

Minimum threads

- Specifies the minimum number of threads for the worker thread pool.
- Corresponds to the ThreadPool.SetMinThreads method.

	Setting name	Values
runtimeconfig.json	`System.Threading.ThreadPool.MinThreads`	An integer that represents the minimum number of threads
MSBuild property	`ThreadPoolMinThreads`	An integer that represents the minimum number of threads
Environment variable	N/A	N/A

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.ThreadPool.MinThreads": 4
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.Threading.ThreadPool.MinThreads": 4
   }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ThreadPoolMinThreads>4</ThreadPoolMinThreads>
  </PropertyGroup>

</Project>
```

Maximum threads

- Specifies the maximum number of threads for the worker thread pool.
- Corresponds to the ThreadPool.SetMaxThreads method.

	Setting name	Values
runtimeconfig.json	`System.Threading.ThreadPool.MaxThreads`	An integer that represents the maximum number of threads
MSBuild property	`ThreadPoolMaxThreads`	An integer that represents the maximum number of threads
Environment variable	N/A	N/A

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.ThreadPool.MaxThreads": 20
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.Threading.ThreadPool.MaxThreads": 20
   }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ThreadPoolMaxThreads>20</ThreadPoolMaxThreads>
  </PropertyGroup>

</Project>
```

Windows thread pool

- For projects on Windows, configures whether thread pool thread management is delegated to the Windows thread pool.
- If you omit this setting or the platform is not Windows, the .NET thread pool is used instead.

- Only applications published with Native AOT on Windows use the Windows thread pool by default, for which you can opt to use the .NET thread pool instead by disabling the config setting.
- The Windows thread pool may perform better in some cases, such as in cases where the minimum number of threads is configured to a high value, or when the Windows thread pool is already being heavily used by the app. There may also be cases where the .NET thread pool performs better, such as in heavy I/O handling on larger machines. It's advisable to check performance metrics when changing this config setting.
- Some APIs are not supported when using the Windows thread pool, such as ThreadPool.SetMinThreads, ThreadPool.SetMaxThreads, and ThreadPool.BindHandle(SafeHandle). Thread pool config settings for minimum and maximum threads are also not effective. An alternative to ThreadPool.BindHandle(SafeHandle) is the ThreadPoolBoundHandle class.

	Setting name	Values	Version introduced
runtimeconfig.json	`System.Threading.ThreadPool.UseWindowsThreadPool`	`true` - enabled `false` - disabled	.NET 8
MSBuild property	`UseWindowsThreadPool`	`true` - enabled `false` - disabled	.NET 8
Environment variable	`DOTNET_ThreadPool_UseWindowsThreadPool`	`1` - enabled `0` - disabled	.NET 8

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.ThreadPool.UseWindowsThreadPool": true
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.Threading.ThreadPool.UseWindowsThreadPool": true
   }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <UseWindowsThreadPool>true</UseWindowsThreadPool>
  </PropertyGroup>

</Project>
```

Thread injection in response to blocking work items

In some cases, the thread pool detects work items that block its threads. To compensate, it injects more threads. In .NET 6+, you can use the following runtime configuration settings to configure thread injection in response to blocking work items. Currently, these settings take effect only for work items that wait for another task to complete, such as in typical sync-over-async cases.

***runtimeconfig.json* setting name**	**Description**	**Version introduced**
`System.Threading.ThreadPool.Blocking.ThreadsToAddWithoutDelay_ProcCountFactor`	After the thread count based on `MinThreads` is reached, this value (after it is multiplied by the processor count) specifies how many additional threads may be created without a delay.	.NET 6
`System.Threading.ThreadPool.Blocking.ThreadsPerDelayStep_ProcCountFactor`	After the thread count based on `ThreadsToAddWithoutDelay` is reached, this value (after it is multiplied by the processor count) specifies after how many threads an additional `DelayStepMs` would be added to the delay before each new thread is created.	.NET 6
`System.Threading.ThreadPool.Blocking.DelayStepMs`	After the thread count based on `ThreadsToAddWithoutDelay` is reached, this value specifies how much additional delay to add per `ThreadsPerDelayStep` threads, which would be applied before each new thread is created.	.NET 6
`System.Threading.ThreadPool.Blocking.MaxDelayMs`	After the thread count based on `ThreadsToAddWithoutDelay` is reached, this value	.NET 6

runtimeconfig.json setting name	Description	Version introduced
	specifies the max delay to use before each new thread is created.	
`System.Threading.ThreadPool.Blocking.IgnoreMemoryUsage`	By default, the rate of thread injection in response to blocking is limited by heuristics that determine whether there is sufficient physical memory available. In some situations, it may be preferable to inject threads more quickly even in low-memory situations. You can disable the memory usage heuristics by turning off this switch.	.NET 7

How the configuration settings take effect

- After the thread count based on `MinThreads` is reached, up to `ThreadsToAddWithoutDelay` additional threads may be created without a delay.
- After that, before each additional thread is created, a delay is induced, starting with `DelayStepMs`.
- For every `ThreadsPerDelayStep` threads that are added with a delay, an additional `DelayStepMs` is added to the delay.
- The delay may not exceed `MaxDelayMs`.
- Delays are only induced before creating threads. If threads are already available, they would be released without delay to compensate for blocking work items.
- Physical memory usage and limits are also used and, beyond a threshold, the system switches to slower thread injection.

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.ThreadPool.Blocking.ThreadsToAddWithoutDelay_ProcCountFactor": 5
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.Threading.ThreadPool.Blocking.ThreadsToAddWithoutDelay_ProcCountFactor": 5
   }
}
```

`AutoreleasePool` for managed threads

This option configures whether each managed thread receives an implicit NSAutoreleasePool when running on a supported macOS platform.

	Setting name	Values	Version introduced
runtimeconfig.json	`System.Threading.Thread.EnableAutoreleasePool`	`true` or `false`	.NET 6
MSBuild property	`AutoreleasePoolSupport`	`true` or `false`	.NET 6
Environment variable	N/A	N/A	N/A

Examples

runtimeconfig.json file:

JSON

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.Thread.EnableAutoreleasePool": true
      }
   }
}
```

runtimeconfig.template.json file:

JSON

```
{
   "configProperties": {
      "System.Threading.Thread.EnableAutoreleasePool": true
   }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <AutoreleasePoolSupport>true</AutoreleasePoolSupport>
  </PropertyGroup>
```

```
</Project>
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Runtime configuration options for WPF

Article • 11/11/2023

This article details the settings you can use to configure Windows Presentation Framework (WPF) in .NET.

> **Note**
>
> .NET 6 standardizes on the prefix `DOTNET_` instead of `COMPlus_` for environment variables that configure .NET run-time behavior. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix for environment variables.

Hardware acceleration in RDP

- Configures whether hardware acceleration is used for WPF apps that are accessed through Remote Desktop Protocol (RDP). Hardware acceleration refers to the use of a computer's graphics processing unit (GPU) to speed up the rendering of graphics and visual effects in an application. This can result in improved performance and more seamless, responsive graphics.
- If you omit this setting, graphics are rendered by software instead. This is equivalent to setting the value to `false`.

Expand table

Setting type	Setting name	Values	Version introduced
runtimeconfig.json	`Switch.System.Windows.Media.EnableHardwareAccelerationInRdp`	`true` - enabled `false` - disabled	.NET 8
Environment variable	N/A		

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see MSBuild properties.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET feedback

.NET is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Troubleshoot app launch failures

Article • 04/01/2023

This article describes some common reasons and possible solutions for application launch failures. It relates to framework-dependent applications, which rely on a .NET installation on your machine.

If you already know which .NET version you need, you can download it from .NET downloads.

.NET installation not found

If a .NET installation isn't found, the application fails to launch with a message similar to:

Console

```
You must install .NET to run this application.

App: C:\repos\myapp\myapp.exe
Architecture: x64
Host version: 7.0.0
.NET location: Not found
```

The error message includes a link to download .NET. You can follow that link to get to the appropriate download page. You can also pick the .NET version (specified by `Host version`) from .NET downloads.

On the download page for the required .NET version, find the **.NET Runtime** download that matches the architecture listed in the error message. You can then install it by downloading and running an **Installer**.

Alternatively, on the download page for the required .NET version, you can download **Binaries** for the specified architecture.

Required framework not found

If a required framework or compatible version isn't found, the application fails to launch with a message similar to:

Console

```
You must install or update .NET to run this application.

App: C:\repos\myapp\myapp.exe
Architecture: x64
Framework: 'Microsoft.NETCore.App', version '5.0.15' (x64)
.NET location: C:\Program Files\dotnet\

The following frameworks were found:
  6.0.2 at [c:\Program Files\dotnet\shared\Microsoft.NETCore.App]
```

The error indicates the name, version, and architecture of the missing framework and the location at which it's expected to be installed. To run the application, you can install a compatible runtime at the specified ".NET location". If the application targets a lower version than one you have installed and you'd like to run it on a higher version, you can also configure roll-forward behavior for the application.

Install a compatible runtime

The error message includes a link to download the missing framework. You can follow this link to get to the appropriate download page.

Alternately, you can download a runtime from the .NET downloads page. There are multiple .NET runtime downloads.

The following table shows the frameworks that each runtime contains.

Expand table

Runtime download	Included frameworks
ASP.NET Core Runtime	Microsoft.NETCore.App Microsoft.AspNetCore.App
.NET Desktop Runtime	Microsoft.NETCore.App Microsoft.WindowsDesktop.App
.NET Runtime	Microsoft.NETCore.App

Select a runtime download that contains the missing framework, and then install it.

On the download page for the required .NET version, find the runtime download that matches the architecture listed in the error message. You likely want to download an **Installer**.

Alternatively, on the download page for the required .NET version, you can download **Binaries** for the specified architecture.

In most cases, when the application that failed to launch is using such an installation, the ".NET location" in the error message points to:

`%ProgramFiles%\dotnet`

Other options

There are other installation and workaround options to consider.

Run the dotnet-install script

Download the dotnet-install script for your operating system. Run the script with options based on the information in the error message. The dotnet-install script reference page shows all available options.

Launch PowerShell and run:

PowerShell

```
dotnet-install.ps1 -Architecture <architecture> -InstallDir <directory> -
Runtime <runtime> -Version <version>
```

For example, the error message in the previous section would correspond to:

PowerShell

```
dotnet-install.ps1 -Architecture x64 -InstallDir "C:\Program Files\dotnet\"
-Runtime dotnet -Version 5.0.15
```

If you encounter an error stating that running scripts is disabled, you may need to set the execution policy to allow the script to run:

PowerShell

```
Set-ExecutionPolicy Bypass -Scope Process
```

For more information on installation using the script, see Install with PowerShell automation.

Download binaries

You can download a binary archive of .NET from the download page. From the **Binaries** column of the runtime download, download the binary release matching the required architecture. Extract the downloaded archive to the ".NET location" specified in the error message.

For more information about manual installation, see Install .NET on Windows

Configure roll-forward behavior

If you already have a higher version of the required framework installed, you can make the application run on that higher version by configuring its roll-forward behavior.

When running the application, you can specify the --roll-forward command line option or set the DOTNET_ROLL_FORWARD environment variable. By default, an application requires a framework that matches the same major version that the application targets, but can use a higher minor or patch version. However, application developers may have specified a different behavior. For more information, see Framework-dependent apps roll-forward.

> **Note**
>
> Since using this option lets the application run on a different framework version than the one for which it was designed, it may result in unintended behavior due to changes between versions of a framework.

Breaking changes

Multi-level lookup disabled for .NET 7 and later

On Windows, before .NET 7, the application could search for frameworks in multiple install locations.

1. Subdirectories relative to:
    - `dotnet` executable when running the application through `dotnet`.
    - `DOTNET_ROOT` environment variable (if set) when running the application through its executable (`apphost`).

2. Globally registered install location (if set) in `HKLM\SOFTWARE\dotnet\Setup\InstalledVersions\<arch>\InstallLocation`.

3. Default install location of `%ProgramFiles%\dotnet` (or `%ProgramFiles(x86)%\dotnet` for 32-bit processes on 64-bit Windows).

This multi-level lookup behavior was enabled by default but could be disabled by setting the environment variable `DOTNET_MULTILEVEL_LOOKUP=0`.

For applications targeting .NET 7 and later, multi-level lookup is completely disabled and only one location—the first location where a .NET installation is found—is searched. When an application is run through `dotnet`, frameworks are only searched for in subdirectories relative to `dotnet`. When an application is run through its executable (`apphost`), frameworks are only searched for in the first of the previously listed locations where .NET is found.

For more information, see Multi-level lookup is disabled.

See also

- Install .NET
- .NET install locations
- Check installed .NET versions
- Framework-dependent applications

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback