Deploy .NET Framework and applications

Article • 05/26/2022

ⓘ Note

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

This article helps you get started deploying .NET Framework with your application. Most of the information is intended for developers, OEMs, and enterprise administrators. Users who want to install .NET Framework on their computers should read Install .NET Framework.

Key Deployment Resources

Use the following links to other MSDN topics for specific information about deploying and servicing the .NET Framework.

Setup and deployment

- General installer and deployment information:
  - Installer options:
    - Web installer
    - Offline installer
  - Installation modes:
    - Silent installation
    - Displaying a UI
  - Reducing system restarts during .NET Framework 4.5 installations
  - Troubleshoot blocked .NET Framework installations and uninstallations
- Deploying the .NET Framework with a client application (for developers):
  - Using InstallShield in a setup and deployment project

    - Using a Visual Studio ClickOnce application
    - Creating a WiX installation package
    - Using a custom installer
    - Additional information for developers
- Deploying the .NET Framework (for OEMs and administrators):
    - Windows Assessment and Deployment Kit (ADK)
    - Administrator's guide

Servicing

- For general information, see the .NET Framework blog.
- Detecting versions
- Detecting service packs and updates

Features That Simplify Deployment

The .NET Framework provides a number of basic features that make it easier to deploy your applications:

- No-impact applications.

  This feature provides application isolation and eliminates DLL conflicts. By default, components do not affect other applications.

- Private components by default.

  By default, components are deployed to the application directory and are visible only to the containing application.

- Controlled code sharing.

  Code sharing requires you to explicitly make code available for sharing instead of being the default behavior.

- Side-by-side versioning.

  Multiple versions of a component or application can coexist, you can choose which versions to use, and the common language runtime enforces versioning policy.

- XCOPY deployment and replication.

  Self-described and self-contained components and applications can be deployed without registry entries or dependencies.

- On-the-fly updates.

  Administrators can use hosts, such as ASP.NET, to update program DLLs, even on remote computers.

- Integration with the Windows Installer.

  Advertisement, publishing, repair, and install-on-demand are all available when deploying your application.

- Enterprise deployment.

  This feature provides easy software distribution, including using Active Directory.

- Downloading and caching.

  Incremental downloads keep downloads smaller, and components can be isolated for use only by the application for low-impact deployment.

- Partially trusted code.

  Identity is based on the code instead of the user, and no certificate dialog boxes appear.

Packaging and Distributing .NET Framework Applications

Some of the packaging and deployment information for the .NET Framework is described in other sections of the documentation. Those sections provide information about the self-describing units called assemblies, which require no registry entries, strong-named assemblies, which ensure name uniqueness and prevent name spoofing, and assembly versioning, which addresses many of the problems associated with DLL conflicts. The following sections provide information about packaging and distributing .NET Framework applications.

Packaging

The .NET Framework provides the following options for packaging applications:

- As a single assembly or as a collection of assemblies.

  With this option, you simply use the .dll or .exe files as they were built.

- As cabinet (CAB) files.

  With this option, you compress files into .cab files to make distribution or download less time consuming.

- As a Windows Installer package or in other installer formats.

  With this option, you create .msi files for use with the Windows Installer, or you package your application for use with some other installer.

Distribution

The .NET Framework provides the following options for distributing applications:

- Use XCOPY or FTP.

  Because common language runtime applications are self-describing and require no registry entries, you can use XCOPY or FTP to simply copy the application to an appropriate directory. The application can then be run from that directory.

- Use code download.

  If you are distributing your application over the Internet or through a corporate intranet, you can simply download the code to a computer and run the application there.

- Use an installer program such as Windows Installer 2.0.

  Windows Installer 2.0 can install, repair, or remove .NET Framework assemblies in the global assembly cache and in private directories.

Installation Location

To determine where to deploy your application's assemblies so they can be found by the runtime, see How the Runtime Locates Assemblies.

Security considerations can also affect how you deploy your application. Security permissions are granted to managed code according to where the code is located. Deploying an application or component to a location where it receives little trust, such as the internet, limits what the application or component can do.

Related Topics

Expand table

Title	Description
How the Runtime Locates Assemblies	Describes how the common language runtime determines which assembly to use to fulfill a binding request.
Best Practices for Assembly Loading	Discusses ways to avoid problems of type identity that can lead to InvalidCastException, MissingMethodException, and other errors.
Reducing System Restarts During .NET Framework 4.5 Installations	Describes the Restart Manager, which prevents reboots whenever possible, and explains how applications that install the .NET Framework can take advantage of it.
Deployment Guide for Administrators	Explains how a system administrator can deploy the .NET Framework and its system dependencies across a network by using Microsoft Endpoint Configuration Manager.
Deployment Guide for Developers	Explains how developers can install .NET Framework on their users' computers with their applications.
Deploying Applications, Services, and Components	Discusses deployment options in Visual Studio, including instructions for publishing an application using the ClickOnce and Windows Installer technologies.
Publishing ClickOnce Applications	Describes how to package a Windows Forms application and deploy it with ClickOnce to client computers on a network.
Package and Deploy resources	Describes the hub and spoke model that the .NET Framework uses to package and deploy resources; covers resource naming conventions, fallback process, and packaging alternatives.
Deploying an Interop Application	Explains how to ship and install interop applications, which typically include a .NET Framework client assembly, one or more interop assemblies representing distinct COM type libraries, and one or more registered COM components.
How to: Get Progress from the .NET Framework 4.5 Installer	Describes how to silently launch and track the .NET Framework setup process while showing your own view of the setup progress.

See also

- Development Guide

Deploying the .NET Framework

Article • 06/04/2024

Note

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

This section of the .NET Framework documentation provides information for developers who want to install the .NET Framework with their applications, and administrators who want to deploy the .NET Framework across a network. It also discusses activation and restart issues associated with deployment, and how to monitor the progress of your .NET Framework installation.

Important

.NET Framework content previously digitally signed using certificates that use the SHA1 algorithm, will be retired in order to support evolving industry standards.

The following versions of .NET Framework will reach end-of-support on *April 26, 2022*: 4.5.2, 4.6, and 4.6.1. After this date, security fixes, updates, and technical support for these versions will no longer be provided.

If you're using .NET Framework 4.5.2, 4.6, or 4.6.1, update your deployed runtime to a more recent version, such as **.NET Framework 4.6.2**, before *April 26, 2022* in order to continue to receive updates and technical support.

Updated SHA2 signed installers will be available for .NET Framework 3.5 SP1, and 4.6.2 through 4.8. For more information, see the **SHA1 retirement plan**, the **.NET 4.5.2, 4.6, and 4.6.1 lifecycle update blog post**, and the **FAQ**.

In This Section

Deployment Guide for Developers Explains how developers can install .NET Framework on their users' computers with their applications.

Deployment Guide for Administrators Explains how a system administrator can deploy the .NET Framework and its system dependencies across a network by using Microsoft Endpoint Configuration Manager.

Reducing System Restarts During .NET Framework 4.5 Installations Describes the Restart Manager, which prevents reboots whenever possible, and explains how applications that install the .NET Framework can take advantage of it.

How to: Get Progress from the .NET Framework 4.5 Installer Describes how to silently launch and track the .NET Framework setup process while showing your own view of the setup progress.

.NET Framework Initialization Errors: Managing the User Experience Explains what happens when a .NET Framework application requires a CLR version that's invalid or not installed on the user's computer, how to resolve these errors, and how to control the error message displayed to the user.

How to: Debug CLR Activation Issues Explains how you can view and debug CLR activation logs to resolve issues you may encounter in getting your application to run with the correct version of the CLR.

See also

- Development Guide

.NET Framework deployment guide for developers

Article • 08/30/2022

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

This article provides information for developers who want to install any version of .NET Framework from .NET Framework 4.5 to .NET Framework 4.8 with their apps.

You can download the redistributable packages and language packs for .NET Framework from the download pages:

- .NET Framework 4.8.1
- .NET Framework 4.8
- .NET Framework 4.7.2
- .NET Framework 4.7.1
- .NET Framework 4.7
- .NET Framework 4.6.2
- .NET Framework 4.6.1
- .NET Framework 4.6
- .NET Framework 4.5.2
- .NET Framework 4.5.1
- .NET Framework 4.5

> **Important**
>
> .NET Framework content previously digitally signed using certificates that use the SHA1 algorithm, will be retired in order to support evolving industry standards.
>
> The following versions of .NET Framework will reach end-of-support on *April 26, 2022*: 4.5.2, 4.6, and 4.6.1. After this date, security fixes, updates, and technical support for these versions will no longer be provided.
>
> If you're using .NET Framework 4.5.2, 4.6, or 4.6.1, update your deployed runtime to a more recent version, such as **.NET Framework 4.6.2**, before *April 26, 2022* in order to continue to receive updates and technical support.

Updated SHA2 signed installers will be available for .NET Framework 3.5 SP1, and 4.6.2 through 4.8. For more information, see the **SHA1 retirement plan**, the **.NET 4.5.2, 4.6, and 4.6.1 lifecycle update blog post**, and the **FAQ**.

Important notes:

- Versions of .NET Framework from .NET Framework 4.5.1 through .NET Framework 4.8 are in-place updates to .NET Framework 4.5, which means they use the same runtime version, but the assembly versions are updated and include new types and members.
- .NET Framework 4.5 and later versions are built incrementally on .NET Framework 4. When you install .NET Framework 4.5 or later versions on a system that has .NET Framework 4 installed, the version 4 assemblies are replaced with newer versions.
- If you are referencing a Microsoft out-of-band package in your app, the assembly will be included in the app package.
- You must have administrator privileges to install .NET Framework 4.5 or later versions.
- .NET Framework 4.5 is included in Windows 8 and Windows Server 2012, so you don't have to deploy it with your app on those operating systems. Similarly, .NET Framework 4.5.1 is included in Windows 8.1 and Windows Server 2012 R2. .NET Framework 4.5.2 isn't included in any operating systems. .NET Framework 4.6 is included in Windows 10, .NET Framework 4.6.1 is included in Windows 10 November Update, and .NET Framework 4.6.2 is included in Windows 10 Anniversary Update. .NET Framework 4.7 is included in Windows 10 Creators Update, .NET Framework 4.7.1 is included in Windows 10 Fall Creators Update, and .NET Framework 4.7.2 is included in Windows 10 October 2018 Update and Windows 10 April 2018 Update. .NET Framework 4.8 is included in Windows 10 May 2019 Update and all later Windows 10 updates. For a full list of hardware and software requirements, see System Requirements.
- Starting with .NET Framework 4.5, your users can view a list of running .NET Framework apps during setup and close them easily. This may help avoid system restarts caused by .NET Framework installations. See Reducing System Restarts.
- Uninstalling .NET Framework 4.5 or later versions also removes pre-existing .NET Framework 4 files. If you want to go back to .NET Framework 4, you must reinstall it and any updates to it. See Installing the .NET Framework 4.

- The .NET Framework 4.5 redistributable was updated on October 9, 2012 to correct an issue related to an improper timestamp on a digital certificate, which caused the digital signature on files produced and signed by Microsoft to expire prematurely. If you previously installed the .NET Framework 4.5 redistributable package dated August 16, 2012, we recommend that you update your copy with the latest redistributable from the .NET Framework download page. For more information about this issue, see Microsoft Security Advisory 2749655.

For information about how a system administrator can deploy the .NET Framework and its system dependencies across a network, see Deployment Guide for Administrators.

Deployment options for your app

When you're ready to publish your app to a web server or other centralized location so that users can install it, you can choose from several deployment methods. Some of these are provided with Visual Studio. The following table lists the deployment options for your app and specifies the .NET Framework redistributable package that supports each option. In addition to these, you can write a custom setup program for your app; for more information, see the section Chaining the .NET Framework Installation to Your App's Setup.

Expand table

Deployment strategy for your app	**Deployment methods available**	**.NET Framework redistributable to use**
Install from the web	- InstallAware - InstallShield - WiX toolset - Manual installation	Web installer
Install from disc	- InstallAware - InstallShield - WiX toolset - Manual installation	Offline installer
Install from a local area network (for enterprise apps)	- ClickOnce	Either web installer (see ClickOnce for restrictions) or offline installer

Redistributable packages

.NET Framework is available in two redistributable packages: web installer (bootstrapper) and offline installer (stand-alone redistributable). All .NET Framework downloads are hosted on the Download .NET Framework page. The following table compares the two packages:

Expand table

	Web installer	Offline installer
Internet connection required?	Yes	No
Size of download	Smaller (includes installer for target platform only)*	Larger*
Language packs	Included**	Must be installed separately, unless you use the package that targets all operating systems
Deployment method	Supports all methods: - ClickOnce - InstallAware - InstallShield - Windows Installer XML (WiX) - Manual installation - Custom setup (chaining)	Supports all methods: - ClickOnce - InstallAware - InstallShield - Windows Installer XML (WiX) - Manual installation - Custom setup (chaining)

* The offline installer is larger because it contains the components for all the target platforms. When you finish running setup, the Windows operating system caches only the installer that was used. If the offline installer is deleted after the installation, the disk space used is the same as that used by the web installer. If the tool you use (for example, InstallAware or InstallShield) to create your app's setup program provides a setup file folder that is removed after installation, the offline installer can be automatically deleted by placing it into the setup folder.

** If you're using the web installer with custom setup, you can use default language settings based on the user's Multilingual User Interface (MUI) setting, or specify another language pack by using the `/LCID` option on the command line. See the section Chaining by Using the Default .NET Framework UI for examples.

Deployment methods

Four deployment methods are available:

- You can set a dependency on .NET Framework. You can specify .NET Framework as a prerequisite in your app's installation, using one of these methods:
  - Use ClickOnce deployment (available with Visual Studio)
  - Create an InstallAware project (free edition available for Visual Studio users)
  - Create an InstallShield project (available with Visual Studio)
  - Use the Windows Installer XML (WiX) toolset
- You can ask your users to install .NET Framework manually.
- You can chain (include) the .NET Framework setup process in your app's setup, and decide how you want to handle the .NET Framework installation experience:
  - Use the default UI. Let the .NET Framework installer provide the installation experience.
  - Customize the UI to present a unified installation experience and to monitor the .NET Framework installation progress.

These deployment methods are discussed in detail in the following sections.

Set a dependency on .NET Framework

If you use ClickOnce, InstallAware, InstallShield, or WiX to deploy your app, you can add a dependency on .NET Framework so it can be installed as part of your app.

ClickOnce deployment

ClickOnce deployment is available for projects that are created with Visual Basic and Visual C#, but it is not available for Visual C++.

In Visual Studio, to choose ClickOnce deployment and add a dependency on .NET Framework:

1. Open the app project you want to publish.
2. In Solution Explorer, open the shortcut menu for your project, and then choose **Properties**.
3. Choose the **Publish** pane.
4. Choose the **Prerequisites** button.

5. In the **Prerequisites** dialog box, make sure that the **Create setup program to install prerequisite components** check box is selected.

6. In the prerequisites list, locate and select the version of .NET Framework that you've used to build your project.

7. Choose an option to specify the source location for the prerequisites, and then choose **OK**.

   If you supply a URL for the .NET Framework download location, you can specify either the .NET Framework download page or a site of your own. If you are placing the redistributable package on your own server, it must be the offline installer and not the web installer. You can only link to the web installer on the .NET Framework download page. The URL can also specify a disc on which your own app is being distributed.

8. In the **Property Pages** dialog box, choose **OK**.

InstallAware deployment

InstallAware builds Windows app (APPX), Windows Installer (MSI), Native Code (EXE), and App-V (Application Virtualization) packages from a single source. Easily include any version of the .NET Framework in your setup, optionally customizing the installation by editing the default scripts. For example, InstallAware pre-installs certificates on Windows 7, without which .NET Framework 4.7 setup fails. For more information on InstallAware, see the InstallAware for Windows Installer website.

InstallShield deployment

InstallShield builds Windows app packages (MSIX, APPX), Windows Installer packages (MSI), and Native Code (EXE) installers. InstallShield also provides Visual Studio integration. For more information, see the InstallShield website.

Windows Installer XML (WiX) deployment

The Windows Installer XML (WiX) toolset builds Windows installation packages from XML source code. WiX supports a command-line environment that can be integrated into your build processes to build MSI and MSM setup packages. By using WiX, you can specify the .NET Framework as a prerequisite, or create a chainer to fully control the .NET Framework deployment experience. For more information about WiX, see the Windows Installer XML (WiX) toolset website.

Install .NET Framework manually

In some situations, it might be impractical to automatically install .NET Framework with your app. In that case, you can have users install .NET Framework themselves. The redistributable package is available in two packages. In your setup process, provide instructions for how users should locate and install .NET Framework.

Chain the .NET Framework installation to your app's setup

If you're creating a custom setup program for your app, you can chain (include) the .NET Framework setup process in your app's setup process. Chaining provides two UI options for the .NET Framework installation:

- Use the default UI provided by the .NET Framework installer.
- Create a custom UI for the .NET Framework installation for consistency with your app's setup program.

Both methods allow you to use either the web installer or the offline installer. Each package has its advantages:

- If you use the web installer, the .NET Framework setup process will decide which installation package is required, and download and install only that package from the web.
- If you use the offline installer, you can include the complete set of .NET Framework installation packages with your redistribution media so that your users don't have to download any additional files from the web during setup.

Chaining by using the default .NET Framework UI

To silently chain the .NET Framework installation process and let the .NET Framework installer provide the UI, add the following command to your setup program:

```
<.NET Framework redistributable> /q /norestart /ChainingPackage <PackageName>
```

For example, if your executable program is Contoso.exe and you want to silently install the .NET Framework 4.5 offline redistributable package, use the command:

```
dotNetFx45_Full_x86_x64.exe /q /norestart /ChainingPackage Contoso
```

You can use additional command-line options to customize the installation. For example:

- To provide a way for users to close running .NET Framework apps to minimize system restarts, set passive mode and use the `/showrmui` option as follows:

  ```
  dotNetFx45_Full_x86_x64.exe /norestart /passive /showrmui /ChainingPackage Contoso
  ```

  This command allows Restart Manager to display a message box that gives users the opportunity to close .NET Framework apps before installing the .NET Framework.

- If you're using the web installer, you can use the `/LCID` option to specify a language pack. For example, to chain the .NET Framework 4.5 web installer to your Contoso setup program and install the Japanese language pack, add the following command to your app's setup process:

  ```
  dotNetFx45_Full_setup.exe /q /norestart /ChainingPackage Contoso /LCID 1041
  ```

  If you omit the `/LCID` option, setup will install the language pack that matches the user's MUI setting.

  Note

  Different language packs may have different release dates. If the language pack you specify is not available at the download center, setup will install the .NET Framework without the language pack. If the .NET Framework is already installed on the user's computer, the setup will install only the language pack.

For a complete list of options, see the Command-Line Options section.

For common return codes, see the Return Codes section.

Chaining by using a Custom UI

If you have a custom setup package, you may want to silently launch and track the .NET Framework setup while showing your own view of the setup progress. If this is the case, make sure that your code covers the following:

- Check for .NET Framework hardware and software requirements.

- Detect whether the correct version of the .NET Framework is already installed on the user's computer.

  > ⓘ **Important**
  >
  > In determining whether the correct version of the .NET Framework is already installed, you should check whether your target version *or* a later version is installed, not whether your target version is installed. In other words, you should evaluate whether the release key you retrieve from the registry is greater than or equal to the release key of your target version, *not* whether it equals the release key of your target version.

- Detect whether the language packs are already installed on the user's computer.
- If you want to control the deployment, silently launch and track the .NET Framework setup process (see How to: Get Progress from the .NET Framework 4.5 Installer).
- If you're deploying the offline installer, chain the language packs separately.
- Customize deployment by using command-line options. For example, if you're chaining the .NET Framework web installer, but you want to override the default language pack, use the `/LCID` option, as described in the previous section.
- Troubleshoot.

Detect .NET Framework

The .NET Framework installer writes registry keys when installation is successful. You can test whether .NET Framework 4.5 or later is installed by checking the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full` folder in the registry for a `DWORD` value named `Release`. (Note that "NET Framework Setup" doesn't begin with a period.) The existence of this key indicates that .NET Framework 4.5 or a later version has been installed on that computer. The value of `Release` indicates which version of .NET Framework is installed.

> ⓘ **Important**
>
> Check for a value **greater than or equal to** the release keyword value when attempting to detect whether a specific version is present.

Expand table

Version	Value of the Release DWORD
.NET Framework 4.8.1	533325
.NET Framework 4.8 installed on Windows 10 May 2020 Update and Windows 10 October 2020 Update	528372
.NET Framework 4.8 installed on Windows 10 May 2019 Update and Windows 10 November 2019 Update	528040
.NET Framework 4.8 installed on all OS versions other than the listed Windows 10 Update versions	528049
.NET Framework 4.7.2 installed on Windows 10 April 2018 Update and on Windows Server, version 1803	461808
.NET Framework 4.7.2 installed on all OS versions other than Windows 10 April 2018 Update, and Windows Server, version 1803. This includes Windows 10 October 2018 Update.	461814
.NET Framework 4.7.1 installed on Windows 10 Fall Creators Update and on Windows Server, version 1709	461308
.NET Framework 4.7.1 installed on all OS versions other than Windows 10 Fall Creators Update and Windows Server, version 1709	461310
.NET Framework 4.7 installed on Windows 10 Creators Update	460798
.NET Framework 4.7 installed on all OS versions other than Windows 10 Creators Update	460805
.NET Framework 4.6.2 installed on Windows 10 Anniversary Edition and on Windows Server 2016	394802
.NET Framework 4.6.2 installed on all OS versions other than Windows 10 Anniversary Edition and Windows Server 2016	394806
.NET Framework 4.6.1 installed on Windows 10 November Update	394254
.NET Framework 4.6.1 installed on all OS versions other than Windows 10 November Update	394271
.NET Framework 4.6 installed on Windows 10	393295
.NET Framework 4.6 installed on all OS versions other than Windows 10	393297
.NET Framework 4.5.2	379893
.NET Framework 4.5.1 installed with Windows 8.1 or Windows Server 2012 R2	378675

Version	Value of the Release DWORD
.NET Framework 4.5.1 installed on Windows 8, Windows 7	378758
.NET Framework 4.5	378389

Detect language packs

You can test whether a specific language pack is installed by checking the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full\LCID folder in the registry for a DWORD value named `Release`. (Note that "NET Framework Setup" doesn't begin with a period.) *LCID* specifies a locale identifier; see supported languages for a list of these.

For example, to detect whether the full Japanese language pack (LCID=1041) is installed, retrieve the following named value from the registry:

Expand table

	Value
Key	HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full\1041
Entry	Release
Type	DWORD

To determine whether the final release version of a language pack is installed for a particular version of .NET Framework from 4.5 through 4.7.2, check the value of the RELEASE key DWORD value described in the previous section, Detecting .NET Framework.

Chaining the language packs to your app setup

.NET Framework provides a set of stand-alone language pack executable files that contain localized resources for specific cultures. The language packs are available from the .NET Framework download pages:

- .NET Framework 4.8.1
- .NET Framework 4.8
- .NET Framework 4.7.2
- .NET Framework 4.7.1
- .NET Framework 4.7

- .NET Framework 4.6.2
- .NET Framework 4.6.1
- .NET Framework 4.6
- .NET Framework 4.5.2
- .NET Framework 4.5.1
- .NET Framework 4.5

Important

The language packs don't contain the .NET Framework components that are required to run an app. You must install .NET Framework by using the web or offline installer before you install a language pack.

Starting with .NET Framework 4.5.1, the package names take the form NDP< `version` >-KB< `number` >-x86-x64-AllOS-< `culture` >.exe, where `version` is the version number of the .NET Framework, `number` is a Microsoft Knowledge Base article number, and `culture` specifies a country/region. An example of one of these packages is `NDP452-KB2901907-x86-x64-AllOS-JPN.exe`. Package names are listed in the Redistributable Packages section earlier in this article.

To install a language pack with the .NET Framework offline installer, you must chain it to your app's setup. For example, to deploy .NET Framework 4.5.1 offline installer with the Japanese language pack, use the following command:

`NDP451-KB2858728-x86-x64-AllOS-JPN.exe /q /norestart /ChainingPackage <ProductName>`

You do not have to chain the language packs if you use the web installer; setup will install the language pack that matches the user's MUI setting. If you want to install a different language, you can use the `/LCID` option to specify a language pack.

For a complete list of command-line options, see the Command-Line Options section.

Troubleshooting

Return codes

The following table lists the most common return codes for the .NET Framework redistributable installer. The return codes are the same for all versions of the installer. For links to detailed information, see the next section.

Expand table

Return code	Description
0	Installation completed successfully.
1602	The user canceled installation.
1603	A fatal error occurred during installation.
1641	A restart is required to complete the installation. This message indicates success.
3010	A restart is required to complete the installation. This message indicates success.
5100	The user's computer does not meet system requirements.

Download error codes

See the following content:

- Background Intelligent Transfer Service (BITS) error codes
- URL moniker error codes
- WinHttp error codes

Other error codes

See the following content:

- Windows Installer error codes
- Windows Update Agent result codes

Uninstall .NET Framework

Starting with Windows 8, you can uninstall .NET Framework 4.5 or later versions by using **Turn Windows features on and off** in Control Panel. In older versions of Windows, you can uninstall .NET Framework 4.5 or later versions by using **Add or Remove Programs** in Control Panel.

Important

For Windows 7 and earlier operating systems, uninstalling .NET Framework 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, or 4.8.1 doesn't restore .NET Framework

4.5 files, and uninstalling .NET Framework 4.5 doesn't restore .NET Framework 4 files. If you want to go back to the older version, you must reinstall it and any updates to it.

Appendix

Command-line options

The following table lists options that you can include when you chain the .NET Framework 4.5 redistributable to your app's setup.

Expand table

Option	Description
/CEIPConsent	Overwrites the default behavior and sends anonymous feedback to Microsoft to improve future deployment experiences. This option can be used only if the setup program prompts for consent and if the user grants permission to send anonymous feedback to Microsoft.
/chainingpackage `packageName`	Specifies the name of the executable that is doing the chaining. This information is sent to Microsoft as anonymous feedback to help improve future deployment experiences. If the package name includes spaces, use double quotation marks as delimiters; for example: **/chainingpackage "Lucerne Publishing"**. For an example of a chaining package, see Getting Progress Information from an Installation Package.
/LCID `LCID` where `LCID` specifies a locale identifier (see supported languages)	Installs the language pack specified by `LCID` and forces the displayed UI to be shown in that language, unless quiet mode is set. For the web installer, this option chain-installs the language package from the web. **Note:** Use this option only with the web installer.
/log `file` | `folder`	Specifies the location of the log file. The default is the temporary folder for the process, and the default file name is based on the package. If the file extension is .txt, a text log is produced. If you specify any other extension or no extension, an HTML log is created.
/msioptions	Specifies options to be passed for .msi and .msp items; for example: `/msioptions "PROPERTY1='Value'"`.

Option	Description
/norestart	Prevents the setup program from rebooting automatically. If you use this option, the chaining app has to capture the return code and handle rebooting (see Getting Progress Information from an Installation Package).
/passive	Sets passive mode. Displays the progress bar to indicate that installation is in progress, but does not display any prompts or error messages to the user. In this mode, when chained by a setup program, the chaining package must handle return codes.
/pipe	Creates a communication channel to enable a chaining package to get progress.
/promptrestart	Passive mode only, if the setup program requires a restart, it prompts the user. This option requires user interaction if a restart is required.
/q	Sets quiet mode.
/repair	Triggers the repair functionality.
/serialdownload	Forces the installation to happen only after the package has been downloaded.
/showfinalerror	Sets passive mode. Displays errors only if the installation is not successful. This option requires user interaction if the installation is not successful.
/showrmui	Used only with the **/passive** option. Displays a message box that prompts users to close .NET Framework apps that are currently running. This message box behaves the same in passive and non-passive mode.
/uninstall	Uninstalls the .NET Framework redistributable.

Supported languages

The following table lists .NET Framework language packs that are available for .NET Framework 4.5 and later versions.

Expand table

LCID	Language – country/region	Culture
1025	Arabic - Saudi Arabia	ar
1028	Chinese – Traditional	zh-Hant

LCID	Language – country/region	Culture
1029	Czech	cs
1030	Danish	da
1031	German – Germany	de
1032	Greek	el
1035	Finnish	fi
1036	French – France	fr
1037	Hebrew	he
1038	Hungarian	hu
1040	Italian – Italy	it
1041	Japanese	ja
1042	Korean	ko
1043	Dutch – Netherlands	nl
1044	Norwegian (Bokmål)	no
1045	Polish	pl
1046	Portuguese – Brazil	pt-BR
1049	Russian	ru
1053	Swedish	sv
1055	Turkish	tr
2052	Chinese – Simplified	zh-Hans
2070	Portuguese – Portugal	pt-PT
3082	Spanish - Spain (Modern Sort)	es

See also

- Deployment Guide for Administrators
- System Requirements
- Install the .NET Framework for developers
- Troubleshoot blocked .NET Framework installations and uninstallations

- Reducing System Restarts During .NET Framework 4.5 Installations
- How to: Get Progress from the .NET Framework 4.5 Installer

.NET Framework Deployment Guide for Administrators

Article • 09/15/2021

Note

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

This step-by-step article describes how a system administrator can deploy .NET Framework 4.5 and its system dependencies across a network by using Microsoft Endpoint Configuration Manager. This article assumes that all target client computers meet the minimum requirements for .NET Framework. For a list of the software and hardware requirements for installing .NET Framework 4.5, see System Requirements.

Note

The software referenced in this document, including, without limitation, .NET Framework 4.5, Configuration Manager, and Active Directory, are each subject to license terms and conditions. These instructions assume that such license terms and conditions have been reviewed and accepted by the appropriate licensees of the software. These instructions do not waive any of the terms and conditions of such license agreements.

For information about support for .NET Framework, see .NET Framework official support policy on the Microsoft Support website.

This topic contains the following sections:

The deployment process

When you have the supporting infrastructure in place, you use Configuration Manager to deploy the .NET Framework redistributable package to computers on the network. Building the infrastructure involves creating and defining five primary areas: collections, a package and program for the software, distribution points, and deployments.

- **Collections** are groups of Configuration Manager resources, such as users, user groups, or computers, to which .NET Framework is deployed. For more information, see Introduction to collections in Configuration Manager in the Configuration Manager documentation library.
- **Packages and programs** typically represent software applications to be installed on a client computer, but they might also contain individual files, updates, or even individual commands. For more information, see Packages and programs in Configuration Manager in the Configuration Manager documentation library.
- **Distribution points** are Configuration Manager site system roles that store files required for software to run on client computers. When the Configuration Manager client receives and processes a software deployment, it contacts a distribution point to download the content associated with the software and to start the installation process. For more information, see Fundamental concepts for content management in Configuration Manager in the Configuration Manager documentation library.
- **Deployments** instruct applicable members of the specified target collection to install the software package.

ⓘ **Important**

The procedures in this topic contain typical settings for creating and deploying a package and program, and might not cover all possible settings. For other Configuration Manager deployment options, see the **Configuration Manager Documentation Library**.

Deploying .NET Framework

You can use Configuration Manager to deploy a silent installation of .NET Framework 4.5, where the users do not interact with the installation process. Follow these steps:

1. Create a collection.

2. Create a package and program for the .NET Framework redistributable.

3. Select a distribution point.

4. Deploy the package.

Create a collection

In this step, you select the computers to which you will deploy the package and program, and group them into a device collection. To create a collection in Configuration Manager, you can use direct membership rules (where you manually specify the collection members) or query rules (where Configuration Manager determines the collection members based on criteria you specify). For more information about membership rules, including queries and direct rules, see Introduction to collections in Configuration Manager in the Configuration Manager Documentation Library.

To create a collection:

1. In the Configuration Manager console, choose **Assets and Compliance**.

2. In the **Assets and Compliance** workspace, choose **Device Collections**.

3. On the **Home** tab in the **Create** group, choose **Create Device Collection**.

4. On the **General** page of the **Create Device Collection Wizard**, enter a name for the collection.

5. Choose **Browse** to specify a limiting collection.

6. On the **Membership Rules** page, choose **Add Rule**, and then choose **Direct Rule** to open the **Create Direct Membership Rule Wizard**. Choose **Next**.

7. On the **Search for Resources** page, in the **Resource class** list, choose **System Resource**. In the **Attribute name** list, choose **Name**. In the **Value** field, enter `%`, and then choose **Next**.

8. On the **Select Resources** page, select the check box for each computer that you want to deploy the .NET Framework to. Choose **Next**, and then complete the wizard.

9. On the **Membership Rules** page of the **Create Device Collection Wizard**, choose **Next**, and then complete the wizard.

Create a package and program for the .NET Framework redistributable package

The following steps create a package for the .NET Framework redistributable manually. The package contains the specified parameters for installing .NET Framework and the location from where the package will be distributed to the target computers.

To create a package:

1. In the Configuration Manager console, choose **Software Library**.
2. In the **Software Library** workspace, expand **Application Management**, and then choose **Packages**.
3. On the **Home** tab, in the **Create** group, choose **Create Package**.
4. On the **Package** page of the **Create Package and Program Wizard**, enter the following information:
   - Name: `.NET Framework 4.5`
   - Manufacturer: `Microsoft`
   - Language. `English (US)`
5. Choose **This package contains source files**, and then choose **Browse** to select the local or network folder that contains the .NET Framework installation files. When you have selected the folder, choose **OK**, and then choose **Next**.
6. On the **Program Type** page of the wizard, choose **Standard Program**, and then choose **Next**.
7. On the **Program** page of the **Create Package and Program Wizard**, enter the following information:

   a. **Name:** `.NET Framework 4.5`

   b. **Command line:** `dotNetFx45_Full_x86_x64.exe /q /norestart /ChainingPackage ADMINDEPLOYMENT` (command-line options are described in the table after these steps)

   c. **Run:** Choose **Hidden**.

   d. **Program can run:** Choose the option that specifies that the program can run regardless of whether a user is logged on.

8. On the **Requirements** page, choose **Next** to accept the default values, and then complete the wizard.

The following table describes the command-line options specified in step 7.

Expand table

Option	Description
/q	Sets quiet mode. No user input is required, and no output is shown.
/norestart	Prevents the Setup program from rebooting automatically. If you use this option, Configuration Manager must handle the computer restart.
/chainingpackage *PackageName*	Specifies the name of the package that is doing the chaining. This information is reported with other installation session information for those who have signed up for the Microsoft Customer Experience Improvement Program (CEIP). If the package name includes spaces, use double quotation marks as delimiters; for example: **/chainingpackage "Chaining Product"**.

These steps create a package named .NET Framework 4.5. The program deploys a silent installation of .NET Framework 4.5. In a silent installation, users do not interact with the installation process, and the chaining application has to capture the return code and handle rebooting; see Getting Progress Information from an Installation Package.

Select a distribution point

To distribute the package and program to client computers from a server, you must first designate a site system as a distribution point and then distribute the package to the distribution point.

Use the following steps to select a distribution point for the .NET Framework 4.5 package you created in the previous section:

1. In the Configuration Manager console, choose **Software Library**.

2. In the **Software Library** workspace, expand **Application Management**, and then choose **Packages**.

3. From the list of packages, select the package **.NET Framework 4.5** that you created in the previous section.

4. On the **Home** tab, in the **Deployment** group, choose **Distribute Content**.

5. On the **General** tab of the **Distribute Content Wizard**, choose **Next**.

6. On the **Content Destination** page of the wizard, choose **Add**, and then choose **Distribution Point**.

7. In the **Add Distribution Points** dialog box, select the distribution point(s) that will host the package and program, and then choose **OK**.

8. Complete the wizard.

The package now contains all the information you need to silently deploy .NET Framework 4.5. Before you deploy the package and program, verify that it was installed on the distribution point; see the "Content status monitoring" section of Monitor content you distribute with Configuration Manager in the Configuration Manager Documentation Library.

Deploy the package

To deploy the .NET Framework 4.5 package and program:

1. In the Configuration Manager console, choose **Software Library**.

2. In the **Software Library** workspace, expand **Application Management**, and then choose **Packages**.

3. From the list of packages, select the package you created named **.NET Framework 4.5**.

4. On the **Home** tab, in the **Deployment** group, choose **Deploy**.

5. On the **General** page of the **Deploy Software Wizard**, choose **Browse**, and then select the collection that you created earlier. Choose **Next**.

6. On the **Content** page of the wizard, verify that the point from which you want to distribute the software is displayed, and then choose **Next**.

7. On the **Deployment Settings** page of the wizard, confirm that **Action** is set to **Install**, and **Purpose** is set to **Required**. This ensures that the software package will be a mandatory installation on the targeted computers. Choose **Next**.

8. On the **Scheduling** page of the wizard, specify when you want .NET Framework to be installed. You can choose **New** to assign an installation time, or instruct the software to install when the user logs on or off, or as soon as possible. Choose **Next**.

9. On the **User Experience** page of the wizard, use the default values and choose **Next**.

> ⚠ **Warning**
>
> Your production environment might have policies that require different selections for the deployment schedule.

10. On the **Distribution Points** page of the wizard, use the default values and choose **Next**.

11. Complete the wizard. You can monitor the progress of the deployment in the **Deployments** node of the **Monitoring** workspace.

The package will now be deployed to the targeted collection and the silent installation of .NET Framework 4.5 will begin. For information about .NET Framework 4.5 installation error codes, see the Return Codes section later in this topic.

Resources

For more information about the infrastructure for testing the deployment of the .NET Framework 4.5 redistributable package, see the following resources.

Active Directory, DNS, DHCP:

- Active Directory Domain Services
- Domain Name System (DNS)
- Dynamic Host Configuration Protocol (DHCP)

SQL Server 2008:

- Installing SQL Server 2008 (SQL Server Video)
- SQL Server 2008 Security Overview for Database Administrators

System Center 2012 Configuration Manager (Management Point, Distribution Point):

- Site Administration for System Center 2012 Configuration Manager

System Center 2012 Configuration Manager client for Windows computers:

- Deploying Clients for System Center 2012 Configuration Manager

Troubleshooting

Log file locations

The following log files are generated during .NET Framework setup:

- %temp%\Microsoft .NET Framework *version**.txt
- %temp%\Microsoft .NET Framework *version**.html

where *version* is the version of .NET Framework that you're installing, such as 4.5 or 4.7.2.

You can also specify the directory to which log files are written by using the `/log` command-line option in the .NET Framework installation command. For more information, see .NET Framework deployment guide for developers.

You can use the log collection tool to collect the .NET Framework log files and to create a compressed cabinet (.cab) file that reduces the size of the files.

Return codes

The following table lists the most common return codes from the .NET Framework 4.5 redistributable installation program. The return codes are the same for all versions of the installer.

For links to detailed information, see the next section, Download error codes.

Expand table

Return code	Description
0	Installation completed successfully.
1602	The user canceled installation.
1603	A fatal error occurred during installation.
1641	A restart is required to complete the installation. This message indicates success.
3010	A restart is required to complete the installation. This message indicates success.
5100	The user's computer does not meet system requirements.

Download error codes

- Background Intelligent Transfer Service (BITS) error codes
- URL moniker error codes

- WinHttp error codes

Other error codes:

- Windows Installer error codes
- Windows Update Agent result codes

See also

- Deployment Guide for Developers
- System Requirements

Reducing System Restarts During .NET Framework 4.5 Installations

Article • 06/04/2024

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

The .NET Framework 4.5 installer uses the Restart Manager to prevent system restarts whenever possible during installation. If your app setup program installs the .NET Framework, it can interface with the Restart Manager to take advantage of this feature. For more information, see How to: Get Progress from the .NET Framework 4.5 Installer.

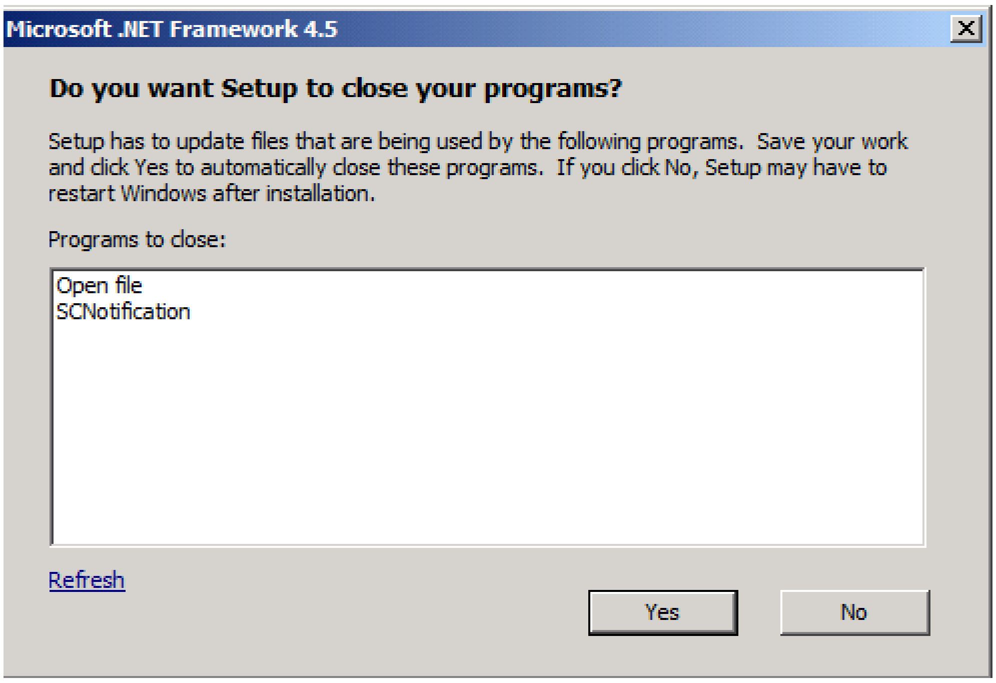
Reasons for a Restart

The .NET Framework 4.5 installation requires a system restart if a .NET Framework 4 app is in use during the installation. This is because the .NET Framework 4.5 replaces .NET Framework 4 files and requires those files to be available during installation. In many cases, the restart can be prevented by preemptively detecting and closing.NET Framework 4 apps that are in use. However, some system apps should not be closed. In these cases, a restart cannot be avoided.

End-User Experience

An end-user who is doing a full installation of the .NET Framework 4.5 is given the opportunity to avoid a system restart if the installer detects .NET Framework 4 apps in use. A message lists all running .NET Framework 4 apps and provides the option to close these apps before the installation. If the user confirms, these apps are shut down by the installer, and a system restart is avoided. If the user does not respond to the message within a certain amount of time, the installation continues without closing any apps.

If the Restart Manager detects a situation that will require a system restart even if running apps are closed, the message is not displayed.


Microsoft .NET Framework 4.5

Do you want Setup to close your programs?

Setup has to update files that are being used by the following programs. Save your work and click Yes to automatically close these programs. If you click No, Setup may have to restart Windows after installation.

Programs to close:

Open file
SCNotification

Using a Chained Installer

If you want to redistribute the .NET Framework with your app, but you want to use your own setup program and UI, you can include (chain) the .NET Framework setup process to your setup process. For more information about chained installations, see Deployment Guide for Developers. To reduce system restarts in chained installations, the .NET Framework installer supplies your setup program with the list of apps to close. Your setup program must provide this information to the user through a user interface such as a message box, get the user's response, and then pass the response back to the .NET Framework installer. For an example of a chained installer, see the article How to: Get Progress from the .NET Framework 4.5 Installer.

If you're using a chained installer, but you do not want to provide your own message box for closing apps, you can use the `/showrmui` and `/passive` options on the command line when you chain the .NET Framework setup process. When you use these options together, the installer shows the message box for closing apps if they can be closed to avoid a system restart. This message box behaves the same in passive mode as it does under the full user interface. See Deployment Guide for Developers for the complete set of command-line options for the .NET Framework redistributable.

See also

- Deployment
- Deployment Guide for Developers
- How to: Get Progress from the .NET Framework 4.5 Installer

How to: Get Progress from the .NET Framework 4.5 Installer

Article • 07/23/2022

The .NET Framework 4.5 is a redistributable runtime. If you develop apps for this version of the .NET Framework, you can include (chain) .NET Framework 4.5 setup as a prerequisite part of your app's setup. To present a customized or unified setup experience, you may want to silently launch .NET Framework 4.5 setup and track its progress while showing your app's setup progress. To enable silent tracking, .NET Framework 4.5 setup (which can be watched) defines a protocol by using a memory-mapped I/O (MMIO) segment to communicate with your setup (the watcher or chainer). This protocol defines a way for a chainer to obtain progress information, get detailed results, respond to messages, and cancel the .NET Framework 4.5 setup.

- **Invocation**. To call .NET Framework 4.5 setup and receive progress information from the MMIO section, your setup program must do the following:

    1. Call the .NET Framework 4.5 redistributable program:

       `dotNetFx45_Full_x86_x64.exe /q /norestart /pipe section-name`

       Where *section name* is any name you want to use to identify your app. .NET Framework setup reads and writes to the MMIO section asynchronously, so you might find it convenient to use events and messages during that time. In the example, the .NET Framework setup process is created by a constructor that both allocates the MMIO section (`TheSectionName`) and defines an event (`TheEventName`):

       ```C++
       Server():ChainerSample::MmioChainer(L"TheSectionName",
       L"TheEventName")
       ```

       Please replace those names with names that are unique to your setup program.

    2. Read from the MMIO section. In .NET Framework 4.5, the download and installation operations are simultaneous: One part of the .NET Framework might be installing while another part is downloading. As a result, progress is sent back (that is, written) to the MMIO section as two numbers

(`m_downloadSoFar` and `m_installSoFar`) that increase from 0 to 255. When 255 is written and the .NET Framework exits, the installation is complete.

- **Exit codes**. The following exit codes from the command to call the .NET Framework 4.5 redistributable program indicate whether setup has succeeded or failed:
  - 0 - Setup completed successfully.
  - 3010 – Setup completed successfully; a system restart is required.
  - 1602 – Setup has been canceled.
  - All other codes - Setup encountered errors; examine the log files created in %temp% for details.
- **Canceling setup**. You can cancel setup at any time by using the `Abort` method to set the `m_downloadAbort` and `m_ installAbort` flags in the MMIO section.

Chainer Sample

The Chainer sample silently launches and tracks .NET Framework 4.5 setup while showing progress. This sample is similar to the Chainer sample provided for the .NET Framework 4. However, in addition, it can avoid system restarts by processing the message box for closing .NET Framework 4 apps. For information about this message box, see Reducing System Restarts During .NET Framework 4.5 Installations. You can use this sample with the .NET Framework 4 installer; in that scenario, the message is simply not sent.

⚠ **Warning**

You must run the example as an administrator.

The following sections describe the significant files in this sample: MMIOChainer.h, ChainingdotNet4.cpp, and IProgressObserver.h.

MMIOChainer.h

- The MMIOChainer.h file contains the data structure definition and the base class from which the chainer class should be derived. The .NET Framework 4.5 extends the MMIO data structure to handle data that the .NET Framework 4.5 installer needs. The changes to the MMIO structure are backward-compatible, so a .NET Framework 4 chainer can work with .NET Framework 4.5 setup without requiring recompilation. However, this scenario does not support the feature for reducing system restarts.

A version field provides a means of identifying revisions to the structure and message format. The .NET Framework setup determines the version of the chainer interface by calling the `VirtualQuery` function to determine the size of the file mapping. If the size is large enough to accommodate the version field, .NET Framework setup uses the specified value. If the file mapping is too small to contain a version field, which is the case with the .NET Framework 4, the setup process assumes version 0 (4). If the chainer does not support the version of the message that .NET Framework setup wants to send, .NET Framework setup assumes an ignore response.

The MMIO data structure is defined as follows:

```C++
// MMIO data structure for interprocess communication
    struct MmioDataStructure
    {
        bool m_downloadFinished;                  // Is download complete?
        bool m_installFinished;                   // Is installation
complete?
        bool m_downloadAbort;                     // Set to cause
downloader to abort.
        bool m_installAbort;                      // Set to cause
installer to abort.
        HRESULT m_hrDownloadFinished;             // Resulting HRESULT for
download.
        HRESULT m_hrInstallFinished;              // Resulting HRESULT for
installation.
        HRESULT m_hrInternalError;
        WCHAR m_szCurrentItemStep[MAX_PATH];
        unsigned char m_downloadSoFar;            // Download progress 0-
255 (0-100% done).
        unsigned char m_installSoFar;             // Installation progress
0-255 (0-100% done).
        WCHAR m_szEventName[MAX_PATH];            // Event that chainer
creates and chainee opens to sync communications.

        BYTE m_version;                           // Version of the data
structure, set by chainer:
                                                  // 0x0: .NET Framework 4
                                                  // 0x1: .NET Framework
4.5

        DWORD m_messageCode;                      // Current message sent
by the chainee; 0 if no message is active.
        DWORD m_messageResponse;                  // Chainer's response to
current message; 0 if not yet handled.
```

```
        DWORD m_messageDataLength;             // Length of the
m_messageData field, in bytes.
        BYTE m_messageData[1];                 // Variable-length
buffer; content depends on m_messageCode.
    };
```

- The `MmioDataStructure` data structure should not be used directly; use the `MmioChainer` class instead to implement your chainer. Derive from the `MmioChainer` class to chain the .NET Framework 4.5 redistributable.

IProgressObserver.h

- The IProgressObserver.h file implements a progress observer. This observer gets notified of download and installation progress (specified as an unsigned `char`, 0-255, indicating 1%-100% complete). The observer is also notified when the chainee sends a message, and the observer should send a response.

```C++
    class IProgressObserver
    {
    public:
        virtual void OnProgress(unsigned char) = 0; // 0 - 255:  255 ==
100%
        virtual void Finished(HRESULT) = 0;         // Called when
operation is complete
        virtual DWORD Send(DWORD dwMessage, LPVOID pData, DWORD
dwDataLength) = 0; // Called when a message is sent
    };
```

ChainingdotNet4.5.cpp

- The ChainingdotNet4.5.cpp file implements the `Server` class, which derives from the `MmioChainer` class and overrides the appropriate methods to display progress information. The MmioChainer creates a section with the specified section name and initializes the chainer with the specified event name. The event name is saved in the mapped data structure. You should make the section and event names unique. The `Server` class in the following code launches the specified setup program, monitors its progress, and returns an exit code.

```C++
class Server : public ChainerSample::MmioChainer, public
ChainerSample::IProgressObserver
```

```
{
public:
    …………….
    Server():ChainerSample::MmioChainer(L"TheSectionName",
L"TheEventName") //customize for your event names
    {}
```

The installation is started in the Main method.

```C++
// Main entry point for program
int __cdecl main(int argc, _In_count_(argc) char **_argv)
{
    int result = 0;
    CString args;
    if (argc > 1)
    {
        args = CString(_argv[1]);
    }

    if (IsNetFx4Present(NETFX45_RC_REVISION))
    {
        printf(".NET Framework 4.5 is already installed");
    }
    else
    {
        result = Server().Launch(args);
    }

    return result;
}
```

- Before launching the installation, the chainer checks to see if the .NET Framework 4.5 is already installed by calling `IsNetFx4Present`:

```C++
///  Checks for presence of the .NET Framework 4.
///    A value of 0 for dwMinimumRelease indicates a check for the .NET
Framework 4 full
///    Any other value indicates a check for a specific compatible
release of the .NET Framework 4.
#define NETFX40_FULL_REVISION 0
// TODO: Replace with released revision number
#define NETFX45_RC_REVISION MAKELONG(50309, 5)   // .NET Framework 4.5
bool IsNetFx4Present(DWORD dwMinimumRelease)
{
    DWORD dwError = ERROR_SUCCESS;
    HKEY hKey = NULL;
    DWORD dwData = 0;
    DWORD dwType = 0;
    DWORD dwSize = sizeof(dwData);

    dwError = ::RegOpenKeyExW(HKEY_LOCAL_MACHINE,
L"SOFTWARE\\Microsoft\\NET Framework Setup\\NDP\\v4\\Full", 0,
KEY_READ, &hKey);
    if (ERROR_SUCCESS == dwError)
    {
        dwError = ::RegQueryValueExW(hKey, L"Release", 0, &dwType,
(LPBYTE)&dwData, &dwSize);

        if ((ERROR_SUCCESS == dwError) && (REG_DWORD != dwType))
        {
            dwError = ERROR_INVALID_DATA;
        }
        else if (ERROR_FILE_NOT_FOUND == dwError)
        {
            // Release value was not found, let's check for 4.0.
            dwError = ::RegQueryValueExW(hKey, L"Install", 0, &dwType,
(LPBYTE)&dwData, &dwSize);

            // Install = (REG_DWORD)1;
            if ((ERROR_SUCCESS == dwError) && (REG_DWORD == dwType) &&
(dwData == 1))
            {
                // treat 4.0 as Release = 0
                dwData = 0;
            }
            else
            {
                dwError = ERROR_INVALID_DATA;
            }
        }
    }

    if (hKey != NULL)
    {
        ::RegCloseKey(hKey);
    }

    return ((ERROR_SUCCESS == dwError) && (dwData >=
dwMinimumRelease));
}
```

- You can change the path of the executable (Setup.exe in the example) in the `Launch` method to point to its correct location, or customize the code to determine the location. The `MmioChainer` base class provides a blocking `Run()` method that the derived class calls.

```
C++
bool Launch(const CString& args)
{
CString cmdline = L"dotNetFx45_Full_x86_x64.exe -pipe TheSectionName "
+ args; // Customize with name and location of setup .exe that you want
to run
STARTUPINFO si = {0};
si.cb = sizeof(si);
PROCESS_INFORMATION pi = {0};

// Launch the Setup.exe that installs the .NET Framework 4.5
BOOL bLaunchedSetup = ::CreateProcess(NULL,
 cmdline.GetBuffer(),
 NULL, NULL, FALSE, 0, NULL, NULL,
 &si,
 &pi);

// If successful
if (bLaunchedSetup != 0)
{
IProgressObserver& observer = dynamic_cast<IProgressObserver&>(*this);
Run(pi.hProcess, observer);

……………………..
return (bLaunchedSetup != 0);
}
```

- The `Send` method intercepts and processes the messages. In this version of the .NET Framework, the only supported message is the close application message.

C++

```
        // SendMessage
        //
        // Send a message and wait for the response.
        // dwMessage: Message to send
        // pData: The buffer to copy the data to
        // dwDataLength: Initially a pointer to the size of pBuffer.
Upon successful call, the number of bytes copied to pBuffer.
        //-------------------------------------------------------------
-
    virtual DWORD Send(DWORD dwMessage, LPVOID pData, DWORD
dwDataLength)
    {
        DWORD dwResult = 0;
        printf("received message: %d\n", dwMessage);
        // Handle message
        switch (dwMessage)
        {
        case MMIO_CLOSE_APPS:
            {
                printf("    applications are holding files in use:\n");
                IronMan::MmioCloseApplications* applications =
reinterpret_cast<IronMan::MmioCloseApplications*>(pData);
                for(DWORD i = 0; i < applications-
>m_dwApplicationsSize; i++)
                {
                    printf("      %ls (%d)\n", applications-
>m_applications[i].m_szName, applications->m_applications[i].m_dwPid);
                }

                printf("    should applications be closed? (Y)es, (N)o,
(R)efresh : ");
                while (dwResult == 0)
                {
                    switch (toupper(getwchar()))
                    {
                    case 'Y':
                        dwResult = IDYES;  // Close apps
                        break;
                    case 'N':
                        dwResult = IDNO;
                        break;
                    case 'R':
                        dwResult = IDRETRY;
                        break;
                    }
                }
                printf("\n");
                break;
            }
        default:
            break;
        }
        printf("  response: %d\n  ", dwResult);
        return dwResult;
    }
};
```

- Progress data is an unsigned `char` between 0 (0%) and 255 (100%).

```
C++

private: // IProgressObserver
    virtual void OnProgress(unsigned char ubProgressSoFar)
    {…………
   }
```

- The HRESULT is passed to the `Finished` method.

```
C++

virtual void Finished(HRESULT hr)
{
```

```
// This HRESULT is communicated over MMIO and may be different than
process
// Exit code of the Chainee Setup.exe itself
printf("\r\nFinished HRESULT: 0x%08X\r\n", hr);
}
```

ⓘ **Important**

The .NET Framework 4.5 redistributable typically writes many progress messages and a single message that indicates completion (on the chainer side). It also reads asynchronously, looking for `Abort` records. If it receives an `Abort` record, it cancels the installation, and writes a finished record with E_ABORT as its data after the installation has been aborted and setup operations have been rolled back.

A typical server creates a random MMIO file name, creates the file (as shown in the previous code example, in `Server::CreateSection`), and launches the redistributable by using the `CreateProcess` method and passing the pipe name with the `-pipe someFileSectionName` option. The server should implement `OnProgress`, `Send`, and `Finished` methods with application UI-specific code.

See also

- Deployment Guide for Developers
- Deployment

.NET Framework initialization errors: Managing the user experience

Article • 07/23/2022

ⓘ Note

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

The common language runtime (CLR) activation system determines the version of the CLR that will be used to run managed application code. In some cases, the activation system might not be able to find a version of the CLR to load. This situation typically occurs when an application requires a CLR version that is invalid or not installed on a given computer. If the requested version is not found, the CLR activation system returns an HRESULT error code from the function or interface that was called, and may display an error message to the user who is running the application. This article provides a list of HRESULT codes and explains how you can prevent the error message from being displayed.

The CLR provides logging infrastructure to help you debug CLR activation issues, as described in How to: Debug CLR Activation Issues. This infrastructure should not be confused with assembly binding logs, which are entirely different.

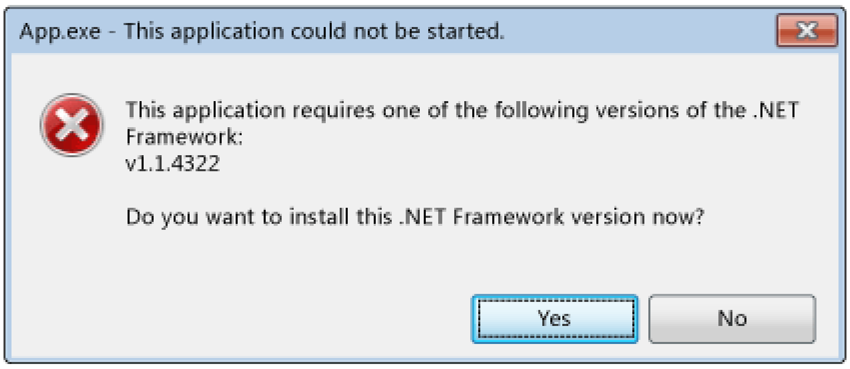
CLR activation HRESULT codes

The CLR activation APIs return HRESULT codes to report the result of an activation operation to a host. CLR hosts should always consult these return values before proceeding with additional operations.

- CLR_E_SHIM_RUNTIMELOAD
- CLR_E_SHIM_RUNTIMEEXPORT
- CLR_E_SHIM_INSTALLROOT
- CLR_E_SHIM_INSTALLCOMP
- CLR_E_SHIM_LEGACYRUNTIMEALREADYBOUND
- CLR_E_SHIM_SHUTDOWNINPROGRESS

UI for initialization errors

If the CLR activation system cannot load the correct version of the runtime that is required by an application, it displays an error message to users to inform them that their computer is not properly configured to run the application, and provides them with an opportunity to remedy the situation. The following error message is typically presented in this situation. The user can choose **Yes** to go to a Microsoft website where they can download the correct .NET Framework version for the application.


App.exe - This application could not be started.

This application requires one of the following versions of the .NET Framework:
v1.1.4322

Do you want to install this .NET Framework version now?

Resolving the initialization error

As a developer, you have a variety of options for controlling the .NET Framework initialization error message. For example, you can use an API flag to prevent the message from being displayed, as discussed in the next section. However, you still have to resolve the issue that prevented your application from loading the requested runtime. Otherwise, your application may not run at all, or some functionality may not be available.

To resolve the underlying issues and provide the best user experience (fewer error messages), we recommend the following:

- For .NET Framework 3.5 (and earlier) applications: Configure your application to support the .NET Framework 4 or later versions (see instructions).
- For .NET Framework 4 applications: Install the .NET Framework 4 redistributable package as part of your application setup. See Deployment Guide for Developers.

Controlling the error message

Displaying an error message to communicate that a requested .NET Framework version was not found can be viewed as either a helpful service or a minor annoyance to users. In either case, you can control this UI by passing flags to the activation APIs.

The ICLRMetaHostPolicy::GetRequestedRuntime method accepts a METAHOST_POLICY_FLAGS enumeration member as input. You can include the METAHOST_POLICY_SHOW_ERROR_DIALOG flag to request an error message if the requested version of the CLR is not found. By default, the error message is not displayed. (The ICLRMetaHost::GetRuntime method does not accept this flag, and does not provide any other way to display the error message.)

Windows provides a SetErrorMode function that you can use to declare whether you want error messages to be shown as a result of code that runs within your process. You can specify the SEM_FAILCRITICALERRORS flag to prevent the error message from being displayed.

However, in some scenarios, it is important to override the SEM_FAILCRITICALERRORS setting set by an application process. For example, if you have a native COM component that hosts the CLR and that is hosted in a process where SEM_FAILCRITICALERRORS is set, you may want to override the flag, depending on the impact of displaying error messages within that particular application process. In this case, you can use one of the following flags to override SEM_FAILCRITICALERRORS:

- Use METAHOST_POLICY_IGNORE_ERROR_MODE with the ICLRMetaHostPolicy::GetRequestedRuntime method.
- Use RUNTIME_INFO_IGNORE_ERROR_MODE with the GetRequestedRuntimeInfo function.

UI policy for CLR-provided hosts

The CLR includes a set of hosts for a variety of scenarios, and these hosts all display an error message when they encounter problems loading the required version of the runtime. The following table provides a list of hosts and their error message policies.

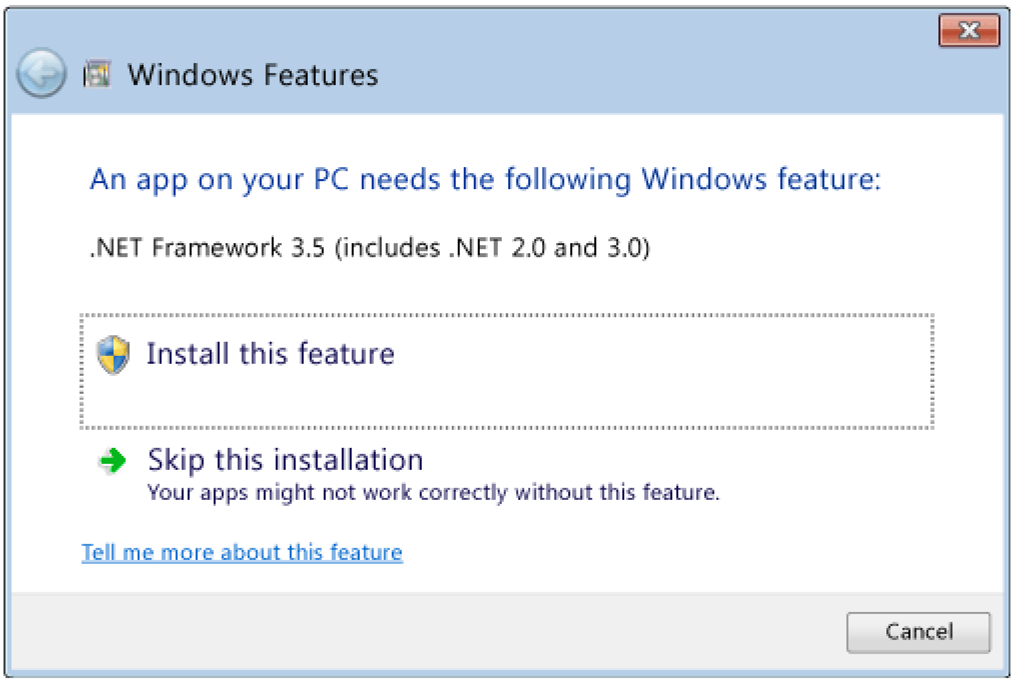
Expand table

CLR host	Description	Error message policy	Can error message be disabled?
Managed EXE host	Launches managed EXEs.	Is shown in case of a missing .NET Framework version	No
Managed COM host	Loads managed COM components into a process.	Is shown in case of a missing .NET Framework version	Yes, by setting the SEM_FAILCRITICALERRORS flag

CLR host	Description	Error message policy	Can error message be disabled?
ClickOnce host	Launches ClickOnce applications.	Is shown in case of a missing .NET Framework version, starting with the .NET Framework 4.5	No
XBAP host	Launches WPF XBAP applications.	Is shown in case of a missing .NET Framework version, starting with the .NET Framework 4.5	No

Windows 8 behavior and UI

The CLR activation system provides the same behavior and UI on Windows 8 as it does on other versions of the Windows operating system, except when it encounters issues loading CLR 2.0. Windows 8 includes .NET Framework 4.5, which uses CLR 4.5. However, Windows 8 does not include .NET Framework 2.0, 3.0, or 3.5, which all use CLR 2.0. As a result, applications that depend on CLR 2.0 do not run on Windows 8 by default. Instead, they display the following dialog box to enable users to install .NET Framework 3.5. Users can also enable the .NET Framework 3.5 in Control Panel. Both options are discussed in the article Install the .NET Framework 3.5 on Windows 11, Windows 10, Windows 8.1, and Windows 8.



An app on your PC needs the following Windows feature:

.NET Framework 3.5 (includes .NET 2.0 and 3.0)

Your apps might not work correctly without this feature.

ⓘ Note

The .NET Framework 4.5 replaces the .NET Framework 4 (CLR 4) on the user's computer. Therefore, .NET Framework 4 applications run seamlessly, without displaying this dialog box, on Windows 8.

When the .NET Framework 3.5 is installed, users can run applications that depend on .NET Framework 2.0, 3.0, or 3.5 on their Windows 8 computers. They can also run .NET Framework 1.0 and 1.1 applications, provided that those applications are not explicitly configured to run only on the .NET Framework 1.0 or 1.1. See Migrating from the .NET Framework 1.1.

Starting with .NET Framework 4.5, CLR activation logging has been improved to include log entries that record when and why the initialization error message is displayed. For more information, see How to: Debug CLR Activation Issues.

See also

- Deployment Guide for Developers
- How to: Configure an app to support .NET Framework 4 or later versions
- How to: Debug CLR Activation Issues
- Install the .NET Framework 3.5 on Windows 11, Windows 10, Windows 8.1, and Windows 8

How to debug CLR activation issues

Article • 09/15/2021

ⓘ **Note**

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

If you encounter problems in getting your application to run with the correct version of the common language runtime (CLR), you can view and debug CLR activation logs. These logs can be very useful in determining the root cause of an activation issue, when your application either loads a different CLR version than expected or doesn't load the CLR at all. The .NET Framework Initialization Errors: Managing the User Experience discusses the experience when no CLR is found for an application.

CLR activation logging can be enabled system-wide by using an HKEY_LOCAL_MACHINE registry key or a system environment variable. The log will be generated until the registry entry or the environment variable is removed. Alternatively, you can use a user or process-local environment variable to enable logging with a different scope and duration.

CLR activation logs shouldn't be confused with assembly binding logs, which are entirely different.

To enable CLR activation logging

Using the registry

1. In the Registry Editor, navigate to HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework (on a 32-bit computer) or HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\.NETFramework folder (on a 64-bit computer).

2. Add a string value named `CLRLoadLogDir`, and set it to the full path of an existing directory where you'd like to store CLR activation logs.

Activation logging remains enabled until you remove the string value.

Using an environment variable

- Set the `COMPLUS_CLRLoadLogDir` environment variable to a string that represents the full path of an existing directory where you'd like to store CLR activation logs.

  How you set the environment variable determines its scope:

  - If you set it at the system level, activation logging is enabled for all .NET Framework applications on that computer until the environment variable is removed.

  - If you set it at the user level, activation logging is enabled only for the current user account. Logging continues until the environment variable is removed.

  - If you set it from within the process before loading the CLR, activation logging is enabled until the process terminates.

  - If you set it at a command prompt before you run an application, activation logging is enabled for any application that is run from that command prompt.

    For example, to store activation logs in the c:\clrloadlogs directory with process-level scope, open a Command Prompt window and type the following before you run the application:

    Console

    ```
    set COMPLUS_CLRLoadLogDir=c:\clrloadlogs
    ```

Example

CLR activation logs provide a large amount of data about CLR activation and the use of the CLR hosting APIs. Most of this data is used internally by Microsoft, but some of the data can also be useful to developers, as described in this article.

The log reflects the order in which the CLR hosting APIs were called. It also includes useful data about the set of installed runtimes detected on the computer. The CLR activation log format is not itself documented, but can be used to aid developers who need to resolve CLR activation issues.

ⓘ **Note**

You cannot open an activation log until the process that uses the CLR has terminated.

ⓘ **Note**

CLR activation logs are not localized; they are always generated in the English language.

In the following example of an activation log, the most useful information is highlighted and described after the log.

Output

```
532,205950.367,CLR Loading log for C:\Tests\myapp.exe
532,205950.367,Log started at 4:26:12 PM on 10/6/2011
532,205950.367,-----------------------------------
532,205950.382,FunctionCall: _CorExeMain
532,205950.382,FunctionCall: ClrCreateInstance, Clsid: {2EBCD49A-1B47-4A61-
B13A-4A03701E594B}, Iid: {E2190695-77B2-492E-8E14-C4B3A7FDD593}
532,205950.382,MethodCall: ICLRMetaHostPolicy::GetRequestedRuntime. Version:
(null), Metahost Policy Flags: 0x168, Binary: (null), Iid: {BD39D1D2-BA2F-
486A-89B0-B4B0CB466891}
532,205950.382,Installed Runtime: v4.0.30319. VERSION_ARCHITECTURE: 0
532,205950.382,Input values for ComputeVersionString follow this line
532,205950.382,-----------------------------------
532,205950.382,Default Application Name: C:\Tests\myapp.exe
532,205950.382,IsLegacyBind is: 0
532,205950.382,IsCapped is 0
532,205950.382,SkuCheckFlags are 0
532,205950.382,ShouldEmulateExeLaunch is 0
532,205950.382,LegacyBindRequired is 0
532,205950.382,-----------------------------------
532,205950.382,Parsing config file: C:\Tests\myapp.exe
532,205950.382,UseLegacyV2RuntimeActivationPolicy is set to 0
532,205950.382,LegacyFunctionCall: GetFileVersion. Filename:
C:\Tests\myapp.exe
532,205950.382,LegacyFunctionCall: GetFileVersion. Filename:
C:\Tests\myapp.exe
532,205950.382,C:\Tests\myapp.exe was built with version: v2.0.50727
532,205950.382,ERROR: Version v2.0.50727 is not present on the machine.
532,205950.398,SEM_FAILCRITICALERRORS is set to 0
532,205950.398,Launching feature-on-demand installation. CmdLine:
C:\Windows\system32\fondue.exe /enable-feature:NetFx3
532,205950.398,FunctionCall: RealDllMain. Reason: 0
532,205950.398,FunctionCall: OnShimDllMainCalled. Reason: 0
```

- **CLR Loading log** provides the path to the executable that started the process that loaded managed code. Note that this could be a native host.

Output

```
532,205950.367,CLR Loading log for C:\Tests\myapp.exe
```

- **Installed Runtime** is the set of CLR versions installed on the computer that are candidates for the activation request.

Output

```
532,205950.382,Installed Runtime: v4.0.30319. VERSION_ARCHITECTURE: 0
```

- **built with version** is the version of the CLR that was used to build the binary that was provided to a method such as ICLRMetaHostPolicy::GetRequestedRuntime.

Output

```
532,205950.382,C:\Tests\myapp.exe was built with version: v2.0.50727
```

- **feature-on-demand installation** refers to enabling the .NET Framework 3.5 on Windows 8. See .NET Framework Initialization Errors: Managing the User Experience for more information about this scenario.

Output

```
532,205950.398,Launching feature-on-demand installation. CmdLine:
C:\Windows\system32\fondue.exe /enable-feature:NetFx3
```

See also

- Deployment
- How to: Configure an app to support .NET Framework 4 or later versions

Deploying .NET Framework Applications

Article • 06/04/2024

Note

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

This section of the .NET Framework documentation provides essential information for deploying .NET Framework applications, including guidelines for loading assemblies, resolving assembly references, and improving the performance of your application through native image generation.

In This Section

How the Runtime Locates Assemblies Describes how the common language runtime locates and binds to the assemblies that make up your application.

Best Practices for Assembly Loading Discusses ways to avoid problems of type identity that can lead to InvalidCastException, MissingMethodException, and other errors.

See also

- Development Guide

How the Runtime Locates Assemblies

Article • 03/30/2023

ⓘ **Note**

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

To successfully deploy your .NET Framework application, you must understand how the common language runtime locates and binds to the assemblies that make up your application. By default, the runtime attempts to bind with the exact version of an assembly that the application was built with. This default behavior can be overridden by configuration file settings.

The common language runtime performs a number of steps when attempting to locate an assembly and resolve an assembly reference. Each step is explained in the following sections. The term probing is often used when describing how the runtime locates assemblies; it refers to the set of heuristics used to locate the assembly based on its name and culture.

ⓘ **Note**

You can view binding information in the log file using the **Assembly Binding Log Viewer (Fuslogvw.exe)**, which is included in the Windows SDK.

Initiating the Bind

The process of locating and binding to an assembly begins when the runtime attempts to resolve a reference to another assembly. This reference can be either static or dynamic. The compiler records static references in the assembly manifest's metadata at build time. Dynamic references are constructed on the fly as a result of calling various methods, such as Assembly.Load.

The preferred way to reference an assembly is to use a full reference, including the assembly name, version, culture, and public key token (if one exists). The runtime uses this information to locate the assembly, following the steps described later in this section. The runtime uses the same resolution process regardless of whether the reference is for a static or dynamic assembly.

You can also make a dynamic reference to an assembly by providing the calling method with only partial information about the assembly, such as specifying only the assembly name. In this case, only the application directory is searched for the assembly, and no other checking occurs. You make a partial reference using any of the various methods for loading assemblies such as Assembly.Load or AppDomain.Load.

Finally, you can make a dynamic reference using a method such as Assembly.Load and provide only partial information; you then qualify the reference using the <qualifyAssembly> element in the application configuration file. This element allows you to provide the full reference information (name, version, culture and, if applicable, the public key token) in your application configuration file instead of in your code. You would use this technique if you wanted to fully qualify a reference to an assembly outside the application directory, or if you wanted to reference an assembly in the global assembly cache but you wanted the convenience of specifying the full reference in the configuration file instead of in your code.

> **Note**
>
> This type of partial reference should not be used with assemblies that are shared among several applications. Because configuration settings are applied per application and not per assembly, a shared assembly using this type of partial reference would require each application using the shared assembly to have the qualifying information in its configuration file.

The runtime uses the following steps to resolve an assembly reference:

1. Determines the correct assembly version by examining applicable configuration files, including the application configuration file, publisher policy file, and machine configuration file. If the configuration file is located on a remote machine, the runtime must locate and download the application configuration file first.

2. Checks whether the assembly name has been bound to before and, if so, uses the previously loaded assembly. If a previous request to load the assembly failed, the request is failed immediately without attempting to load the assembly.

   > **Note**
   >
   > The caching of assembly binding failures is new in .NET Framework version 2.0.

3. Checks the global assembly cache. If the assembly is found there, the runtime uses this assembly.

4. Probes for the assembly using the following steps:

   a. If configuration and publisher policy do not affect the original reference and if the bind request was created using the Assembly.LoadFrom method, the runtime checks for location hints.

   b. If a codebase is found in the configuration files, the runtime checks only this location. If this probe fails, the runtime determines that the binding request failed and no other probing occurs.

   c. Probes for the assembly using the heuristics described in the probing section. If the assembly is not found after probing, the runtime requests the Windows Installer to provide the assembly. This acts as an install-on-demand feature.

   > **Note**
   >
   > There is no version checking for assemblies without strong names, nor does the runtime check in the global assembly cache for assemblies without strong names.

Step 1: Examining the Configuration Files

Assembly binding behavior can be configured at different levels based on three XML files:

- Application configuration file.
- Publisher policy file.
- Machine configuration file.

These files follow the same syntax and provide information such as binding redirects, the location of code, and binding modes for particular assemblies. Each configuration file can contain an <assemblyBinding> element that redirects the binding process. The child elements of the <assemblyBinding> element include the <dependentAssembly> element. The children of <dependentAssembly> element include the <assemblyIdentity> element, the <bindingRedirect> element, and the <codeBase> element.

ⓘ Note

Configuration information can be found in the three configuration files; not all elements are valid in all configuration files. For example, binding mode and private path information can only be in the application configuration file. For a complete list of the information that is contained in each file, see Configuring Apps by Using Configuration Files.

Application Configuration File

First, the common language runtime checks the application configuration file for information that overrides the version information stored in the calling assembly's manifest. The application configuration file can be deployed with an application, but is not required for application execution. Usually the retrieval of this file is almost instantaneous, but in situations where the application base is on a remote computer, such as in a Web-based scenario, the configuration file must be downloaded.

For client executables, the application configuration file resides in the same directory as the application's executable and has the same base name as the executable with a .config extension. For example, the configuration file for C:\Program Files\Myapp\Myapp.exe is C:\Program Files\Myapp\Myapp.exe.config. In a browser-based scenario, the HTML file must use the **<link>** element to explicitly point to the configuration file.

The following code provides a simple example of an application configuration file. This example adds a TextWriterTraceListener to the Listeners collection to enable recording debug information to a file.

XML

```xml
<configuration>
   <system.diagnostics>
      <trace useGlobalLock="false" autoflush="true" indentsize="0">
         <listeners>
            <add name="myListener"
type="System.Diagnostics.TextWriterTraceListener, system version=1.0.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
initializeData="c:\myListener.log" />
         </listeners>
      </trace>
   </system.diagnostics>
</configuration>
```

Publisher Policy File

Second, the runtime examines the publisher policy file, if one exists. Publisher policy files are distributed by a component publisher as a fix or update to a shared component. These files contain compatibility information issued by the publisher of the shared component that directs an assembly reference to a new version. Unlike application and machine configuration files, publisher policy files are contained in their own assembly that must be installed in the global assembly cache.

The following is an example of a Publisher Policy configuration file:

XML

```xml
<configuration>
    <runtime>
        <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">

            <dependentAssembly>
                <assemblyIdentity name="asm6"
publicKeyToken="c0305c36380ba429" />
                <bindingRedirect oldVersion="3.0.0.0" newVersion="2.0.0.0"/>
            </dependentAssembly>

        </assemblyBinding>
    </runtime>
</configuration>
```

To create an assembly, you can use the Al.exe (Assembly Linker) tool with a command such as the following:

Console

```console
Al.exe /link:asm6.exe.config /out:policy.3.0.asm6.dll /keyfile:
compatkey.dat /v:3.0.0.0
```

`compatkey.dat` is a strong-name key file. This command creates a strong-named assembly you can place in the global assembly cache.

> **Note**
>
> Publisher policy affects all applications that use a shared component.

The publisher policy configuration file overrides version information that comes from the application (that is, from the assembly manifest or from the application configuration file). If there is no statement in the application configuration file to redirect the version specified in the assembly manifest, the publisher policy file overrides the version specified in the assembly manifest. However, if there is a redirecting statement in the application configuration file, publisher policy overrides that version rather than the one specified in the manifest.

A publisher policy file is used when a shared component is updated and the new version of the shared component should be picked up by all applications using that component. The settings in the publisher policy file override settings in the application configuration file, unless the application configuration file enforces safe mode.

Safe Mode

Publisher policy files are usually explicitly installed as part of a service pack or program update. If there is any problem with the upgraded shared component, you can ignore the overrides in the publisher policy file using safe mode. Safe mode is determined by the **<publisherPolicy apply="yes|no"/>** element, located only in the application configuration file. It specifies whether the publisher policy configuration information should be removed from the binding process.

Safe mode can be set for the entire application or for selected assemblies. That is, you can turn off the policy for all assemblies that make up the application, or turn it on for some assemblies but not others. To selectively apply publisher policy to assemblies that make up an application, set **<publisherPolicy apply=no/>** and specify which assemblies you want to be affected using the <**dependentAssembly**> element. To apply publisher policy to all assemblies that make up the application, set **<publisherPolicy apply=no/>** with no dependent assembly elements. For more about configuration, see Configuring Apps by using Configuration Files.

Machine Configuration File

Third, the runtime examines the machine configuration file. This file, called Machine.config, resides on the local computer in the Config subdirectory of the root directory where the runtime is installed. This file can be used by administrators to specify assembly binding restrictions that are local to that computer. The settings in the machine configuration file take precedence over all other configuration settings; however, this does not mean that all configuration settings should be put in this file. The version determined by the administrator policy file is final, and cannot be overridden. Overrides specified in the Machine.config file affect all applications. For more information about configuration files, see Configuring Apps by using Configuration Files.

Step 2: Checking for Previously Referenced Assemblies

If the requested assembly has also been requested in previous calls, the common language runtime uses the assembly that is already loaded. This can have ramifications when naming assemblies that make up an application. For more information about naming assemblies, see Assembly Names.

If a previous request for the assembly failed, subsequent requests for the assembly are failed immediately without attempting to load the assembly. Starting with .NET Framework version 2.0, assembly binding failures are cached, and the cached information is used to determine whether to attempt to load the assembly.

> ⓘ **Note**
>
> To revert to the behavior of the .NET Framework versions 1.0 and 1.1, which did not cache binding failures, include the <disableCachingBindingFailures> Element in your configuration file.

Step 3: Checking the Global Assembly Cache

For strong-named assemblies, the binding process continues by looking in the global assembly cache. The global assembly cache stores assemblies that can be used by several applications on a computer. All assemblies in the global assembly cache must have strong names.

Step 4: Locating the Assembly through Codebases or Probing

After the correct assembly version has been determined by using the information in the calling assembly's reference and in the configuration files, and after it has checked in the global assembly cache (only for strong-named assemblies), the common language runtime attempts to find the assembly. The process of locating an assembly involves the following steps:

1. If a <codeBase> element is found in the application configuration file, the runtime checks the specified location. If a match is found, that assembly is used and no probing occurs. If the assembly is not found there, the binding request fails.

2. The runtime then probes for the referenced assembly using the rules specified later in this section.

> Note
>
> If you have multiple versions of an assembly in a directory and you want to reference a particular version of that assembly, you must use the <codeBase> element instead of the `privatePath` attribute of the <probing> element. If you use the <probing> element, the runtime stops probing the first time it finds an assembly that matches the simple assembly name referenced, whether it is a correct match or not. If it is a correct match, that assembly is used. If it is not a correct match, probing stops and binding fails.

Locating the Assembly through Codebases

Codebase information can be provided by using a <codeBase> element in a configuration file. This codebase is always checked before the runtime attempts to probe for the referenced assembly. If a publisher policy file containing the final version redirect also contains a <codeBase> element, that <codeBase> element is the one that is used. For example, if your application configuration file specifies a <codeBase> element, and a publisher policy file that is overriding the application information also specifies a <codeBase> element, the <codeBase> element in the publisher policy file is used.

If no match is found at the location specified by the <codeBase> element, the bind request fails and no further steps are taken. If the runtime determines that an assembly matches the calling assembly's criteria, it uses that assembly. When the file specified by the given <codeBase> element is loaded, the runtime checks to make sure that the name, version, culture, and public key match the calling assembly's reference.

> Note
>
> Referenced assemblies outside the application's root directory must have strong names and must either be installed in the global assembly cache or specified using the <codeBase> element.

Locating the Assembly through Probing

If there is no <codeBase> element in the application configuration file, the runtime probes for the assembly using four criteria:

- Application base, which is the root location where the application is being executed.
- Culture, which is the culture attribute of the assembly being referenced.
- Name, which is the name of the referenced assembly.
- The `privatePath` attribute of the <probing> element, which is the user-defined list of subdirectories under the root location. This location can be specified in the application configuration file and in managed code using the AppDomainSetup.PrivateBinPath property for an application domain. When specified in managed code, the managed code `privatePath` is probed first, followed by the path specified in the application configuration file.

Probing the Application Base and Culture Directories

The runtime always begins probing in the application's base, which can be either a URL or the application's root directory on a computer. If the referenced assembly is not found in the application base and no culture information is provided, the runtime searches any subdirectories with the assembly name. The directories probed include:

- [application base] / [assembly name].dll
- [application base] / [assembly name] / [assembly name].dll

If culture information is specified for the referenced assembly, only the following directories are probed:

- [application base] / [culture] / [assembly name].dll
- [application base] / [culture] / [assembly name] / [assembly name].dll

Probing with the privatePath Attribute

In addition to the culture subdirectories and the subdirectories named for the referenced assembly, the runtime also probes directories specified using the `privatePath` attribute of the <probing> element. The directories specified using the `privatePath` attribute must be subdirectories of the application's root directory. The directories probed vary depending on whether culture information is included in the referenced assembly request.

The runtime stops probing the first time it finds an assembly that matches the simple assembly name referenced, whether it is a correct match or not. If it is a correct match, that assembly is used. If it is not a correct match, probing stops and binding fails.

If culture is included, the following directories are probed:

- [application base] / [binpath] / [culture] / [assembly name].dll
- [application base] / [binpath] / [culture] / [assembly name] / [assembly name].dll

If culture information is not included, the following directories are probed:

- [application base] / [binpath] / [assembly name].dll
- [application base] / [binpath] / [assembly name] / [assembly name].dll

Probing Examples

Given the following information:

- Referenced assembly name: myAssembly
- Application root directory: `http://www.code.microsoft.com`
- <probing> element in configuration file specifies: bin
- Culture: de

The runtime probes the following URLs:

- `http://www.code.microsoft.com/de/myAssembly.dll`
- `http://www.code.microsoft.com/de/myAssembly/myAssembly.dll`
- `http://www.code.microsoft.com/bin/de/myAssembly.dll`
- `http://www.code.microsoft.com/bin/de/myAssembly/myAssembly.dll`

Multiple Assemblies with the Same Name

The following example shows how to configure multiple assemblies with the same name.

XML

```xml
<dependentAssembly>
   <assemblyIdentity name="Server" publicKeyToken="c0305c36380ba429" />
   <codeBase version="1.0.0.0" href="v1/Server.dll" />
```

```
    <codeBase version="2.0.0.0" href="v2/Server.dll" />
</dependentAssembly>
```

Other Locations Probed

Assembly location can also be determined using the current binding context. This most often occurs when the Assembly.LoadFrom method is used and in COM interop scenarios. If an assembly uses the LoadFrom method to reference another assembly, the calling assembly's location is considered to be a hint about where to find the referenced assembly. If a match is found, that assembly is loaded. If no match is found, the runtime continues with its search semantics and then queries the Windows Installer to provide the assembly. If no assembly is provided that matches the binding request, an exception is thrown. This exception is a TypeLoadException in managed code if a type was referenced, or a FileNotFoundException if an assembly being loaded was not found.

For example, if Assembly1 references Assembly2 and Assembly1 was downloaded from `http://www.code.microsoft.com/utils`, that location is considered to be a hint about where to find Assembly2.dll. The runtime then probes for the assembly in `http://www.code.microsoft.com/utils/Assembly2.dll` and `http://www.code.microsoft.com/utils/Assembly2/Assembly2.dll`. If Assembly2 is not found at either of those locations, the runtime queries the Windows Installer.

See also

- Best Practices for Assembly Loading
- Deployment

Best Practices for Assembly Loading

Article • 09/15/2021

ⓘ **Note**

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

This article discusses ways to avoid problems of type identity that can lead to InvalidCastException, MissingMethodException, and other errors. The article discusses the following recommendations:

- Understand the advantages and disadvantages of load contexts
- Avoid binding on partial assembly names
- Avoid loading an assembly into multiple contexts
- Avoid loading multiple versions of an assembly into the same context
- Consider switching to the default load context

The first recommendation, understand the advantages and disadvantages of load contexts, provides background information for the other recommendations, because they all depend on a knowledge of load contexts.

Understand the Advantages and Disadvantages of Load Contexts

Within an application domain, assemblies can be loaded into one of three contexts, or they can be loaded without context:

- The default load context contains assemblies found by probing the global assembly cache, the host assembly store if the runtime is hosted (for example, in SQL Server), and the ApplicationBase and PrivateBinPath of the application domain. Most overloads of the Load method load assemblies into this context.
- The load-from context contains assemblies that are loaded from locations that are not searched by the loader. For example, add-ins might be installed in a directory that is not under the application path. Assembly.LoadFrom,

AppDomain.CreateInstanceFrom, and AppDomain.ExecuteAssembly are examples of methods that load by path.

- The reflection-only context contains assemblies loaded with the ReflectionOnlyLoad and ReflectionOnlyLoadFrom methods. Code in this context cannot be executed, so it is not discussed here. For more information, see How to: Load Assemblies into the Reflection-Only Context.

- If you generated a transient dynamic assembly by using reflection emit, the assembly is not in any context. In addition, most assemblies that are loaded by using the LoadFile method are loaded without context, and assemblies that are loaded from byte arrays are loaded without context unless their identity (after policy is applied) establishes that they are in the global assembly cache.

The execution contexts have advantages and disadvantages, as discussed in the following sections.

Default Load Context

When assemblies are loaded into the default load context, their dependencies are loaded automatically. Dependencies that are loaded into the default load context are found automatically for assemblies in the default load context or the load-from context. Loading by assembly identity increases the stability of applications by ensuring that unknown versions of assemblies are not used (see the Avoid Binding on Partial Assembly Names section).

Using the default load context has the following disadvantages:

- Dependencies that are loaded into other contexts are not available.

- You cannot load assemblies from locations outside the probing path into the default load context.

Load-From Context

The load-from context lets you load an assembly from a path that is not under the application path, and therefore is not included in probing. It enables dependencies to be located and loaded from that path, because the path information is maintained by the context. In addition, assemblies in this context can use dependencies that are loaded into the default load context.

Loading assemblies by using the Assembly.LoadFrom method, or one of the other methods that load by path, has the following disadvantages:

- If an assembly with the same identity is already loaded in the load-from context, LoadFrom returns the loaded assembly even if a different path was specified.
- If an assembly is loaded with LoadFrom, and later an assembly in the default load context tries to load the same assembly by display name, the load attempt fails. This can occur when an assembly is deserialized.
- If an assembly is loaded with LoadFrom, and the probing path includes an assembly with the same identity but in a different location, an InvalidCastException, MissingMethodException, or other unexpected behavior can occur.
- LoadFrom demands FileIOPermissionAccess.Read and FileIOPermissionAccess.PathDiscovery, or WebPermission, on the specified path.
- If a native image exists for the assembly, it is not used.
- The assembly cannot be loaded as domain-neutral.
- In the .NET Framework versions 1.0 and 1.1, policy is not applied.

No Context

Loading without context is the only option for transient assemblies that are generated with reflection emit. Loading without context is the only way to load multiple assemblies that have the same identity into one application domain. The cost of probing is avoided.

Assemblies that are loaded from byte arrays are loaded without context unless the identity of the assembly, which is established when policy is applied, matches the identity of an assembly in the global assembly cache; in that case, the assembly is loaded from the global assembly cache.

Loading assemblies without context has the following disadvantages:

- Other assemblies cannot bind to assemblies that are loaded without context, unless you handle the AppDomain.AssemblyResolve event.
- Dependencies are not loaded automatically. You can preload them without context, preload them into the default load context, or load them by handling the AppDomain.AssemblyResolve event.
- Loading multiple assemblies with the same identity without context can cause type identity problems similar to those caused by loading assemblies with the same identity into multiple contexts. See Avoid Loading an Assembly into Multiple Contexts.

- If a native image exists for the assembly, it is not used.
- The assembly cannot be loaded as domain-neutral.
- In the .NET Framework versions 1.0 and 1.1, policy is not applied.

Avoid Binding on Partial Assembly Names

Partial name binding occurs when you specify only part of the assembly display name (FullName) when you load an assembly. For example, you might call the Assembly.Load method with only the simple name of the assembly, omitting the version, culture, and public key token. Or you might call the Assembly.LoadWithPartialName method, which first calls the Assembly.Load method and, if that fails to locate the assembly, searches the global assembly cache and loads the latest available version of the assembly.

Partial name binding can cause many problems, including the following:

- The Assembly.LoadWithPartialName method might load a different assembly with the same simple name. For example, two applications might install two completely different assemblies that both have the simple name `GraphicsLibrary` into the global assembly cache.
- The assembly that is actually loaded might not be backward-compatible. For example, not specifying the version might result in the loading of a much later version than the version your program was originally written to use. Changes in the later version might cause errors in your application.
- The assembly that is actually loaded might not be forward-compatible. For example, you might have built and tested your application with the latest version of an assembly, but partial binding might load a much earlier version that lacks features your application uses.
- Installing new applications can break existing applications. An application that uses the LoadWithPartialName method can be broken by installing a newer, incompatible version of a shared assembly.
- Unexpected dependency loading can occur. It you load two assemblies that share a dependency, loading them with partial binding might result in one assembly using a component that it was not built or tested with.

Because of the problems it can cause, the LoadWithPartialName method has been marked obsolete. We recommend that you use the Assembly.Load method instead, and specify full assembly display names. See Understand the Advantages and Disadvantages of Load Contexts and Consider Switching to the Default Load Context.

If you want to use the LoadWithPartialName method because it makes assembly loading easy, consider that having your application fail with an error message that identifies the missing assembly is likely to provide a better user experience than automatically using an unknown version of the assembly, which might cause unpredictable behavior and security holes.

Avoid Loading an Assembly into Multiple Contexts

Loading an assembly into multiple contexts can cause type identity problems. If the same type is loaded from the same assembly into two different contexts, it is as if two different types with the same name had been loaded. An InvalidCastException is thrown if you try to cast one type to the other, with the confusing message that type `MyType` cannot be cast to type `MyType`.

For example, suppose that the `ICommunicate` interface is declared in an assembly named `Utility`, which is referenced by your program and also by other assemblies that your program loads. These other assemblies contain types that implement the `ICommunicate` interface, allowing your program to use them.

Now consider what happens when your program is run. Assemblies that are referenced by your program are loaded into the default load context. If you load a target assembly by its identity, using the Load method, it will be in the default load context, and so will its dependencies. Both your program and the target assembly will use the same `Utility` assembly.

However, suppose you load the target assembly by its file path, using the LoadFile method. The assembly is loaded without any context, so its dependencies are not automatically loaded. You might have a handler for the AppDomain.AssemblyResolve event to supply the dependency, and it might load the `Utility` assembly with no context by using the LoadFile method. Now when you create an instance of a type that is contained in the target assembly and try to assign it to a variable of type `ICommunicate`, an InvalidCastException is thrown because the runtime considers the `ICommunicate` interfaces in the two copies of the `Utility` assembly to be different types.

There are many other scenarios in which an assembly can be loaded into multiple contexts. The best approach is to avoid conflicts by relocating the target assembly in your application path and using the Load method with the full display name. The assembly is then loaded into the default load context, and both assemblies use the same `Utility` assembly.

If the target assembly must remain outside your application path, you can use the LoadFrom method to load it into the load-from context. If the target assembly was compiled with a reference to your application's `Utility` assembly, it will use the `Utility` assembly that your application has loaded into the default load context. Note that problems can occur if the target assembly has a dependency on a copy of the `Utility` assembly located outside your application path. If that assembly is loaded into the load-from context before your application loads the `Utility` assembly, your application's load will fail.

The Consider Switching to the Default Load Context section discusses alternatives to using file path loads such as LoadFile and LoadFrom.

Avoid Loading Multiple Versions of an Assembly into the Same Context

Loading multiple versions of an assembly into one load context can cause type identity problems. If the same type is loaded from two versions of the same assembly, it is as if two different types with the same name had been loaded. An InvalidCastException is thrown if you try to cast one type to the other, with the confusing message that type `MyType` cannot be cast to type `MyType`.

For example, your program might load one version of the `Utility` assembly directly, and later it might load another assembly that loads a different version of the `Utility` assembly. Or a coding error might cause two different code paths in your application to load different versions of an assembly.

In the default load context, this problem can occur when you use the Assembly.Load method and specify complete assembly display names that include different version numbers. For assemblies that are loaded without context, the problem can be caused by using the Assembly.LoadFile method to load the same assembly from different paths. The runtime considers two assemblies that are loaded from different paths to be different assemblies, even if their identities are the same.

In addition to type identity problems, multiple versions of an assembly can cause a MissingMethodException if a type that is loaded from one version of the assembly is passed to code that expects that type from a different version. For example, the code might expect a method that was added to the later version.

More subtle errors can occur if the behavior of the type changed between versions. For example, a method might throw an unexpected exception or return an unexpected value.

Carefully review your code to ensure that only one version of an assembly is loaded. You can use the AppDomain.GetAssemblies method to determine which assemblies are loaded at any given time.

Consider Switching to the Default Load Context

Examine your application's assembly loading and deployment patterns. Can you eliminate assemblies that are loaded from byte arrays? Can you move assemblies into the probing path? If assemblies are located in the global assembly cache or in the application domain's probing path (that is, its ApplicationBase and PrivateBinPath), you can load the assembly by its identity.

If it is not possible to put all your assemblies in the probing path, consider alternatives such as using the .NET Framework add-in model, placing assemblies into the global assembly cache, or creating application domains.

Consider Using the .NET Framework Add-In Model

If you are using the load-from context to implement add-ins, which typically are not installed in the application base, use the .NET Framework add-in model. This model provides isolation at the application domain or process level, without requiring you to manage application domains yourself. For information about the add-in model, see Add-ins and Extensibility.

Consider Using the Global Assembly Cache

Place assemblies in the global assembly cache to get the benefit of a shared assembly path that is outside the application base, without losing the advantages of the default load context or taking on the disadvantages of the other contexts.

Consider Using Application Domains

If you determine that some of your assemblies cannot be deployed in the application's probing path, consider creating a new application domain for those assemblies. Use an AppDomainSetup to create the new application domain, and use the AppDomainSetup.ApplicationBase property to specify the path that contains the assemblies you want to load. If you have multiple directories to probe, you can set the

ApplicationBase to a root directory and use the AppDomainSetup.PrivateBinPath property to identify the subdirectories to probe. Alternatively, you can create multiple application domains and set the ApplicationBase of each application domain to the appropriate path for its assemblies.

Note that you can use the Assembly.LoadFrom method to load these assemblies. Because they are now in the probing path, they will be loaded into the default load context instead of the load-from context. However, we recommend that you switch to the Assembly.Load method and supply full assembly display names to ensure that correct versions are always used.

See also

- Assembly.Load
- Assembly.LoadFrom
- Assembly.LoadFile
- AppDomain.AssemblyResolve

.NET Framework Client Profile

Article • 06/04/2024

ⓘ **Note**

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

The .NET Client Profile is a subset of the .NET Framework, which was provided with .NET Framework 4 and earlier versions and was optimized for client applications. The .NET Framework is a development platform for Windows, Windows Phone and Microsoft Azure and provides a managed app execution environment and the .NET Framework class library. The .NET Framework 4 and earlier versions provided two deployment options: the full .NET Framework and the Client Profile. The Client Profile enabled faster deployment and smaller app installation packages than the full .NET Framework.

Starting with .NET Framework 4.5, the Client Profile has been discontinued and only the full redistributable package is available. Optimizations provided by .NET Framework 4.5, such as smaller download size and faster deployment, have eliminated the need for a separate deployment package. The single redistributable streamlines the installation process and simplifies your app's deployment options.

However, if you are targeting the .NET Framework 4 or 3.5 and want to learn more about the Client Profile and when to use it, see .NET Framework Client Profile in the .NET Framework 4 documentation.

When you install .NET Framework 4.5, the .NET Framework 4 Client Profile is updated to the full version of the .NET Framework. For information about installing .NET Framework 4.5, see Install the .NET Framework for developers.

See also

- .NET Framework Client Profile (.NET Framework 4)
- Visual Studio Multi-Targeting Overview
- Troubleshooting .NET Framework Targeting Errors
- How to: Target a Version of the .NET Framework

Side-by-Side Execution in the .NET Framework

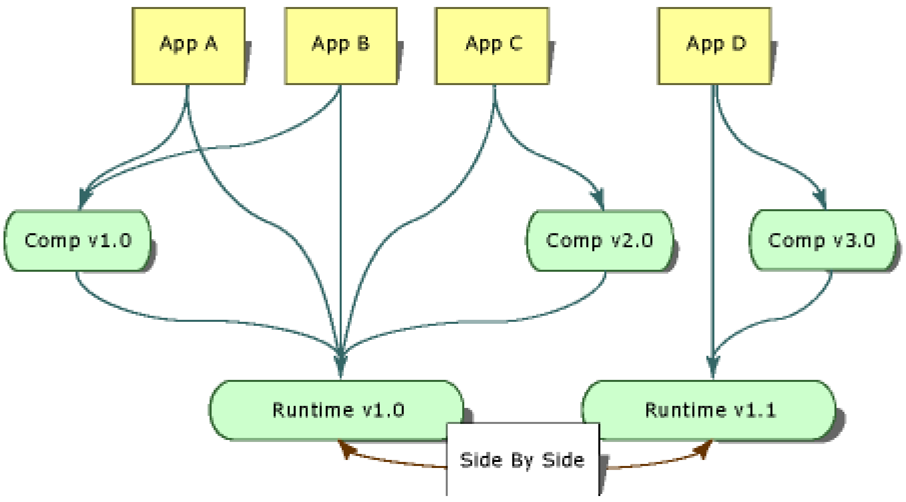
Article • 09/15/2021

ⓘ **Note**

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

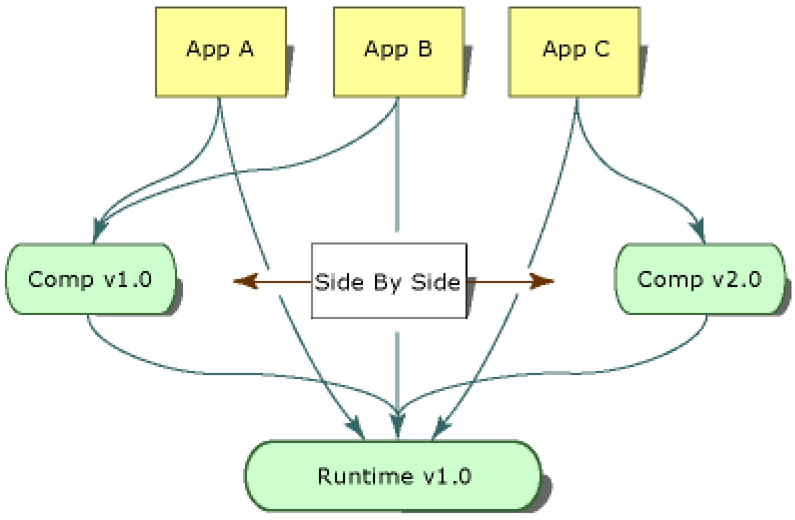
Side-by-side execution is the ability to run multiple versions of an application or component on the same computer. You can have multiple versions of the common language runtime, and multiple versions of applications and components that use a version of the runtime, on the same computer at the same time.

The following illustration shows several applications using two different versions of the runtime on the same computer. Applications A, B, and C use runtime version 1.0, while application D uses runtime version 1.1.



The .NET Framework consists of the common language runtime and a collection of assemblies that contain the API types. The runtime and the .NET Framework assemblies are versioned separately. For example, version 4.0 of the runtime is actually version 4.0.319, while version 1.0 of the .NET Framework assemblies is version 1.0.3300.0.

The following illustration shows several applications using two different versions of a component on the same computer. Application A and B use version 1.0 of the component while Application C uses version 2.0 of the same component.



Side-by-side execution gives you more control over which versions of a component an application binds to, and more control over which version of the runtime an application uses.

Benefits of Side-by-Side Execution

Prior to Windows XP and the .NET Framework, DLL conflicts occurred because applications were unable to distinguish between incompatible versions of the same code. Type information contained in a DLL was bound only to a file name. An application had no way of knowing if the types contained in a DLL were the same types that the application was built with. As a result, a new version of a component could overwrite an older version and break applications.

Side-by-side execution and the .NET Framework provide the following features to eliminate DLL conflicts:

- Strong-named assemblies.

  Side-by-side execution uses strong-named assemblies to bind type information to a specific version of an assembly. This prevents an application or component from binding to an invalid version of an assembly. Strong-named assemblies also allow multiple versions of a file to exist on the same computer and to be used by applications. For more information, see Strong-Named Assemblies.

- Version-aware code storage.

  The .NET Framework provides version-aware code storage in the global assembly cache. The global assembly cache is a computer-wide code cache present on all computers with the .NET Framework installed. It stores assemblies based on version, culture, and publisher information, and supports multiple versions of components and applications. For more information, see Global Assembly Cache.

- Isolation.

  Using the .NET Framework, you can create applications and components that execute in isolation. Isolation is an essential component of side-by-side execution. It involves being aware of the resources you are using and sharing resources with confidence among multiple versions of an application or component. Isolation also includes storing files in a version-specific way. For more information about isolation, see Guidelines for Creating Components for Side-by-Side Execution.

Version Compatibility

Versions 1.0 and 1.1 of the .NET Framework are designed to be compatible with one another. An application built with the .NET Framework version 1.0 should run on version 1.1, and an application built with the .NET Framework version 1.1 should run on version 1.0. Note, however, that API features added in version 1.1 of the .NET Framework will not work with version 1.0 of the .NET Framework. Applications created with version 2.0 will run on version 2.0 only. Version 2.0 applications will not run on version 1.1 or earlier.

Versions of the .NET Framework are treated as a single unit consisting of the runtime and its associated .NET Framework assemblies (a concept referred to as assembly unification). You can redirect assembly binding to include other versions of the .NET Framework assemblies, but overriding the default assembly binding can be risky and must be rigorously tested before deployment.

Locating Runtime Version Information

Information on which runtime version an application or component was compiled with and which versions of the runtime the application requires to run are stored in two locations. When an application or component is compiled, information on the runtime version used to compile it is stored in the managed executable. Information on the runtime versions the application or component requires is stored in the application configuration file.

Runtime Version Information in the Managed Executable

The portable executable (PE) file header of each managed application and component contains information about the runtime version it was built with. The common language runtime uses this information to determine the most likely version of the runtime the application needs to run.

Runtime Version Information in the Application Configuration File

In addition to the information in the PE file header, an application can be deployed with an application configuration file that provides runtime version information. The application configuration file is an XML-based file that is created by the application developer and that ships with an application. The <requiredRuntime> Element of the <startup> section, if it is present in this file, specifies which versions of the runtime and which versions of a component the application supports. You can also use this file in testing to test an application's compatibility with different versions of the runtime.

Unmanaged code, including COM and COM+ applications, can have application configuration files that the runtime uses for interacting with managed code. The application configuration file affects any managed code that you activate through COM. The file can specify which runtime versions it supports, as well as assembly redirects. By default, COM interop applications calling to managed code use the latest version of the runtime installed on the computer.

For more information about the application configuration files, see Configuring Apps.

Determining Which Version of the Runtime to Load

The common language runtime uses the following information to determine which version of the runtime to load for an application:

- The runtime versions that are available.
- The runtime versions that an application supports.

Supported Runtime Versions

The runtime uses the application configuration file and the portable executable (PE) file header to determine which version of the runtime an application supports. If no application configuration file is present, the runtime loads the runtime version specified in the application's PE file header, if that version is available.

If an application configuration file is present, the runtime determines the appropriate runtime version to load based on the results of the following process:

1. The runtime examines the <supportedRuntime> Element element in the application configuration file. If one or more of the supported runtime versions

specified in the <**supportedRuntime**> element are present, the runtime loads the runtime version specified by the first <**supportedRuntime**> element. If this version is not available, the runtime examines the next <**supportedRuntime**> element and attempts to load the runtime version specified. If this runtime version is not available, subsequent <**supportedRuntime**> elements are examined. If none of the supported runtime versions are available, the runtime fails to load a runtime version and displays a message to the user (see step 3).

2. The runtime reads the PE file header of the application's executable file. If the runtime version specified by the PE file header is available, the runtime loads that version. If the runtime version specified is not available, the runtime searches for a runtime version determined by Microsoft to be compatible with the runtime version in the PE header. If that version is not found, the process continues to step 3.

3. The runtime displays a message stating that the runtime version supported by the application is unavailable. The runtime is not loaded.

   > **Note**
   >
   > You can suppress the display of this message by using the NoGuiFromShim value under the registry key HKLM\Software\Microsoft\.NETFramework or using the environment variable COMPLUS_NoGuiFromShim. For example, you can suppress the message for applications that do not typically interact with the user, such as unattended installations or Windows services. When this message display is suppressed, the runtime writes a message to the event log. Set the registry value NoGuiFromShim to 1 to suppress this message for all applications on a computer. Alternately, set the COMPLUS_NoGuiFromShim environment variable to 1 to suppress the message for applications running in a particular user context.

> **Note**
>
> After a runtime version is loaded, assembly binding redirects can specify that a different version of an individual .NET Framework assembly be loaded. These binding redirects affect only the specific assembly that is redirected.

Partially Qualified Assembly Names and Side-by-Side Execution

Because they are a potential source of side-by-side problems, partially qualified assembly references can be used only to bind to assemblies within an application directory. Avoid partially qualified assembly references in your code.

To mitigate partially qualified assembly references in code, you can use the <qualifyAssembly> element in an application configuration file to fully qualify partially qualified assembly references that occur in code. Use the **<qualifyAssembly>** element to specify only fields that were not set in the partial reference. The assembly identity listed in the **fullName** attribute must contain all the information needed to fully qualify the assembly name: assembly name, public key, culture, and version.

The following example shows the application configuration file entry to fully qualify an assembly called `myAssembly`.

XML

```xml
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
<qualifyAssembly partialName="myAssembly"
fullName="myAssembly,
      version=1.0.0.0,
publicKeyToken=...,
      culture=neutral"/>
</assemblyBinding>
```

Whenever an assembly load statement references `myAssembly`, these configuration file settings cause the runtime to automatically translate the partially qualified `myAssembly` reference to a fully qualified reference. For example, Assembly.Load("myAssembly") becomes Assembly.Load("myAssembly, version=1.0.0.0, publicKeyToken=..., culture=neutral").

ⓘ Note

You can use the **LoadWithPartialName** method to bypass the common language runtime restriction that prohibits partially referenced assemblies from being loaded from the global assembly cache. This method should be used only in remoting scenarios as it can easily cause problems in side-by-side execution.

Related Topics

Expand table

Title	Description
How to: Enable and Disable Automatic Binding Redirection	Describes how to bind an application to a specific version of an assembly.
Configuring Assembly Binding Redirection	Explains how to redirect assembly binding references to a specific version of the .NET Framework assemblies.
In-Process Side-by-Side Execution	Discusses how you can use in-process side-by-side runtime host activation to run multiple versions of the CLR in a single process.
Assemblies in .NET	Provides a conceptual overview of assemblies.
Application Domains	Provides a conceptual overview of application domains.

Reference

<supportedRuntime> Element

Configuring Assembly Binding Redirection

Article • 06/04/2024

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

By default, applications use the set of .NET Framework assemblies that shipped with the runtime version used to compile the application. You can use the **appliesTo** attribute on the <assemblyBinding> element in an application configuration file to redirect assembly binding references to a specific version of the .NET Framework assemblies. This optional attribute uses a .NET Framework version number to indicate which version it applies to. If no **appliesTo** attribute is specified, the **<assemblyBinding>** element applies to all versions of the .NET Framework.

The **appliesTo** attribute was introduced in the .NET Framework version 1.1; it is ignored by .NET Framework version 1.0. This means that all **<assemblyBinding>** elements are applied when using .NET Framework version 1.0, even if an **appliesTo** attribute is specified.

> **Note**
>
> Use the **appliesTo** attribute to limit assembly binding redirection to a specific version of the runtime.

For example, to redirect assembly binding for a .NET Framework version 1.0 assembly, you would include the following XML code in your application configuration file.

XML

```xml
<runtime>
        <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1"
appliesTo="v1.0.3705">
            <dependentAssembly>
               * assembly information goes here *
            </dependentAssembly>
       </assemblyBinding>
</runtime>
```

The **<assemblyBinding>** elements are order-sensitive. You should enter assembly binding redirection information for any .NET Framework version 1.0 assemblies first, followed by assembly binding redirection information for any .NET Framework version 1.1 assemblies. Finally, enter assembly binding redirection information for any .NET Framework assembly redirection that does not use the **appliesTo** attribute and therefore applies to all versions of the .NET Framework. In case of a conflict in redirection, the first matching redirection statement in the configuration file is used.

For example, to redirect one reference to a .NET Framework version 1.0 assembly and another reference to a .NET Framework version 1.1 assembly, you would use the pattern shown in the following pseudocode.

```xml
<assemblyBinding xmlns="..." appliesTo="v1.0.3705">
  <!-- .NET Framework version 1.0 redirects here. -->
</assemblyBinding>

<assemblyBinding xmlns="..." appliesTo="v1.1.4322">
  <!-- .NET Framework version 1.1 redirects here. -->
</assemblyBinding>

<assemblyBinding xmlns="...">
  <!-- Redirects meant for all versions of the .NET Framework. -->
</assemblyBinding>
```

Debugging Configuration File Errors

The runtime parses configuration files once when an application domain is created, and loads code into that application domain. The common language runtime handles errors in a configuration file by ignoring the entry. The runtime ignores the entire configuration file if it contains malformed XML. For invalid XML, only the invalid sections are ignored.

You can determine whether a configuration file is being used by determining whether assembly binding redirects are occurring. Use the Assembly Binding Log Viewer (Fuslogvw.exe) to see which assemblies are being loaded. To see all assembly binds, you must set an entry for **ForceLog** in the registry.

See also

- How to: Enable and Disable Automatic Binding Redirection

Guidelines for Creating Components for Side-by-Side Execution

Article • 06/04/2024

> **Note**
>
> This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Follow these general guidelines to create managed applications or components designed for side-by-side execution:

- Bind type identity to a particular version of a file.

  The common language runtime binds type identity to a particular file version by using strong-named assemblies. To create an application or component for side-by-side execution, you must give all assemblies a strong name. This creates precise type identity and ensures that any type resolution is directed to the correct file. A strong-named assembly contains version, culture, and publisher information that the runtime uses to locate the correct file to fulfill a binding request.

- Use version-aware storage.

  The runtime uses the global assembly cache to provide version-aware storage. The global assembly cache is a version-aware directory structure installed on every computer that uses the .NET Framework. Assemblies installed in the global assembly cache are not overwritten when a new version of that assembly is installed.

- Create an application or component that runs in isolation.

  An application or component that runs in isolation must manage resources to avoid conflicts when two instances of the application or component are running simultaneously. The application or component must also use a version-specific file structure.

Application and Component Isolation

One key to successfully designing an application or component for side-by-side execution is isolation. The application or component must manage all resources, particularly file I/O, in an isolated manner. Follow these guidelines to make sure your application or component runs in isolation:

- Write to the registry in a version-specific way. Store values in hives or keys that indicate the version, and do not share information or state across versions of a component. This prevents two applications or components running at the same time from overwriting information.
- Make named kernel objects version-specific so that a race condition does not occur. For example, a race condition occurs when two semaphores from two versions of the same application wait on each other.
- Make file and directory names version-aware. This means that file structures should rely on version information.
- Create user accounts and groups in a version-specific manner. User accounts and groups created by an application should be identified by version. Do not share user accounts and groups between versions of an application.

Installing and Uninstalling Versions

When designing an application for side-by-side execution, follow these guidelines concerning installing and uninstalling versions:

- Do not delete information from the registry that may be needed by other applications running under a different version of the .NET Framework.
- Do not replace information in the registry that may be needed by other applications running under a different version of the .NET Framework.
- Do not unregister COM components that may be needed by other applications running under a different version of the .NET Framework.
- Do not change **InprocServer32** or other registry entries for a COM server that was already registered.
- Do not delete user accounts or groups that may be needed by other applications running under a different version of the .NET Framework.
- Do not add anything to the registry that contains an unversioned path.

File Version Number and Assembly Version Number

File version is a Win32 version resource that is not used by the runtime. In general, you update the file version even for an in-place update. Two identical files can have different file version information, and two different files can have the same file version information.

The assembly version is used by the runtime for assembly binding. Two identical assemblies with different version numbers are treated as two different assemblies by the runtime.

The Global Assembly Cache tool (Gacutil.exe) allows you to replace an assembly when only the file version number is newer. The installer generally does not install over an assembly unless the assembly version number is greater.

See also

- Side-by-Side Execution
- How to: Enable and Disable Automatic Binding Redirection

In-Process Side-by-Side Execution

Article • 07/23/2022

Note

This article is specific to .NET Framework. It doesn't apply to newer implementations of .NET, including .NET 6 and later versions.

Starting with the .NET Framework 4, you can use in-process side-by-side hosting to run multiple versions of the common language runtime (CLR) in a single process. By default, managed COM components run with the .NET Framework version they were built with, regardless of the .NET Framework version that is loaded for the process.

Background

The .NET Framework has always provided side-by-side hosting for managed code applications, but before the .NET Framework 4, it did not provide that functionality for managed COM components. In the past, managed COM components that were loaded into a process ran either with the version of the runtime that was already loaded or with the latest installed version of the .NET Framework. If this version was not compatible with the COM component, the component would fail.

The .NET Framework 4 provides a new approach to side-by-side hosting that ensures the following:

- Installing a new version of the .NET Framework has no effect on existing applications.
- Applications run against the version of the .NET Framework that they were built with. They do not use the new version of the .NET Framework unless expressly directed to do so. However, it is easier for applications to transition to using a new version of the .NET Framework.

Effects on Users and Developers

- **End users and system administrators**. These users can now have greater confidence that when they install a new version of the runtime, either independently or with an application, it will have no impact on their computers. Existing applications will continue to run as they did before.

- **Application developers**. Side-by-side hosting has almost no effect on application developers. By default, applications always run against the version of the .NET Framework they were built on; this has not changed. However, developers can override this behavior and direct the application to run under a newer version of the .NET Framework (see scenario 2).

- **Library developers and consumers**. Side-by-side hosting does not solve the compatibility problems that library developers face. A library that is directly loaded by an application -- either through a direct reference or through an Assembly.Load call -- continues to use the runtime of the AppDomain it is loaded into. You should test your libraries against all versions of the .NET Framework that you want to support. If an application is compiled using the .NET Framework 4 runtime but includes a library that was built using an earlier runtime, that library will use the .NET Framework 4 runtime as well. However, if you have an application that was built using an earlier runtime and a library that was built using the .NET Framework 4, you must force your application to also use the .NET Framework 4 (see scenario 3).

- **Managed COM component developers**. In the past, managed COM components automatically ran using the latest version of the runtime installed on the computer. You can now execute COM components against the version of the runtime they were built with.

  As shown by the following table, components that were built with the .NET Framework version 1.1 can run side by side with version 4 components, but they cannot run with version 2.0, 3.0, or 3.5 components, because side-by-side hosting is not available for those versions.

  Expand table

.NET Framework version	1.1	2.0 - 3.5	4
1.1	Not applicable	No	Yes
2.0 - 3.5	No	Not applicable	Yes
4	Yes	Yes	Not applicable

> ⓘ Note
>
> .NET Framework versions 3.0 and 3.5 are built incrementally on version 2.0, and do not need to run side by side. These are inherently the same version.

Common Side-by-Side Hosting Scenarios

- **Scenario 1:** Native application that uses COM components built with earlier versions of the .NET Framework.

  .NET Framework versions installed: The .NET Framework 4 and all other versions of the .NET Framework used by the COM components.

  What to do: In this scenario, do nothing. The COM components will run with the version of the .NET Framework they were registered with.

- **Scenario 2**: Managed application built with the .NET Framework 2.0 SP1 that you would prefer to run with .NET Framework 2.0, but are willing to run on the .NET Framework 4 if version 2.0 is not present.

  .NET Framework versions installed: An earlier version of the .NET Framework and the .NET Framework 4.

  What to do: In the application configuration file in the application directory, use the <startup> element and the <supportedRuntime> element set as follows:

  XML

  ```xml
  <configuration>
    <startup >
      <supportedRuntime version="v2.0.50727" />
      <supportedRuntime version="v4.0" />
    </startup>
  </configuration>
  ```

- **Scenario 3:** Native application that uses COM components built with earlier versions of the .NET Framework that you want to run with the .NET Framework 4.

  .NET Framework versions installed: The .NET Framework 4.

  What to do: In the application configuration file in the application directory, use the `<startup>` element with the `useLegacyV2RuntimeActivationPolicy` attribute set to `true` and the `<supportedRuntime>` element set as follows:

  XML

  ```xml
  <configuration>
    <startup useLegacyV2RuntimeActivationPolicy="true">
      <supportedRuntime version="v4.0" />
    </startup>
  </configuration>
  ```

Example

The following example demonstrates an unmanaged COM host that is running a managed COM component by using the version of the .NET Framework that the component was compiled to use.

To run the following example, compile and register the following managed COM component using .NET Framework 3.5. To register the component, on the **Project** menu, click **Properties**, click the **Build** tab, and then select the **Register for COM interop** check box.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;

namespace BasicComObject
{
    [ComVisible(true), Guid("9C99C4B5-CA54-4c58-8988-49B6811BA53B")]
    public class MyObject : SimpleObjectModel.IPrintInfo
    {
        public MyObject()
        {
        }
        public void PrintInfo()
        {
            Console.WriteLine("MyObject was activated in {0} runtime 
in:\n\tAppDomain {1}:{2}", 
System.Runtime.InteropServices.RuntimeEnvironment.GetSystemVersion(), 
AppDomain.CurrentDomain.Id, AppDomain.CurrentDomain.FriendlyName);
        }
    }
}
```

Compile the following unmanaged C++ application, which activates the COM object that is created by the previous example.

C++

```
#include "stdafx.h"
#include <string>
#include <iostream>
#include <objbase.h>
#include <string.h>
#include <process.h>
```

```
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    char input;
    CoInitialize(NULL) ;
    CLSID clsid;
    HRESULT hr;
    HRESULT clsidhr = CLSIDFromString(L"{9C99C4B5-CA54-4c58-8988-
49B6811BA53B}",&clsid);
    hr = -1;
    if (FAILED(clsidhr))
    {
        printf("Failed to construct CLSID from String\n");
    }
    UUID id = __uuidof(IUnknown);
    IUnknown * pUnk = NULL;
    hr = ::CoCreateInstance(clsid,NULL,CLSCTX_INPROC_SERVER,id,(void **)
&pUnk);
    if (FAILED(hr))
    {
        printf("Failed CoCreateInstance\n");
    }else
    {
        pUnk->AddRef();
        printf("Succeeded\n");
    }

    DISPID dispid;
    IDispatch* pPrintInfo;
    pUnk->QueryInterface(IID_IDispatch, (void**)&pPrintInfo);
    OLECHAR FAR* szMethod[1];
    szMethod[0]=OLESTR("PrintInfo");
    hr = pPrintInfo->GetIDsOfNames(IID_NULL,szMethod, 1,
LOCALE_SYSTEM_DEFAULT, &dispid);
    DISPPARAMS dispparams;
    dispparams.cNamedArgs = 0;
    dispparams.cArgs = 0;
    VARIANTARG* pvarg = NULL;
    EXCEPINFO * pexcepinfo = NULL;
    WORD wFlags = DISPATCH_METHOD ;
;
    LPVARIANT pvRet = NULL;
    UINT * pnArgErr = NULL;
    hr = pPrintInfo->Invoke(dispid,IID_NULL, LOCALE_USER_DEFAULT, wFlags,
        &dispparams, pvRet, pexcepinfo, pnArgErr);
    printf("Press Enter to exit");
    scanf_s("%c",&input);
    CoUninitialize();
    return 0;
}
```

See also

- <startup> Element
- <supportedRuntime> Element