Configure apps by using configuration files

Article • 03/05/2025

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

.NET Framework gives developers and administrators control and flexibility over the way applications run through *configuration files*. Configuration files are XML files that can be changed as needed. An administrator can control which protected resources an application can access, which versions of assemblies an application will use, and where remote applications and objects are located. Developers can put settings in configuration files, eliminating the need to recompile an application every time a setting changes. This section describes what can be configured and why configuring an application might be useful.

> **Note**
>
> Managed code can use the classes in the System.Configuration namespace to read settings from the configuration files, but not to write settings to those files.

This article describes the syntax of configuration files and provides information about the three types of configuration files: machine, application, and security.

Configuration file format

Configuration files contain elements, which are logical data structures that set configuration information. Within a configuration file, you use tags to mark the beginning and end of an element. For example, the `<runtime>` element consists of `<runtime>` *child elements* `</runtime>`. An empty element would be written as `<runtime/>` or `<runtime></runtime>`.

As with all XML files, the syntax in configuration files is case-sensitive.

You specify configuration settings using predefined attributes, which are name/value pairs inside an element's start tag. The following example specifies two attributes

(`version` and `href`) for the `<codeBase>` element, which specifies where the runtime can locate an assembly (for more information, see Specifying an Assembly's Location).

XML

```
<codeBase version="2.0.0.0"
          href="http://www.litwareinc.com/myAssembly.dll"/>
```

Machine configuration files

The machine configuration file, *Machine.config*, contains settings that apply to an entire computer. This file is located in the %*runtime install path*%\Config directory. *Machine.config* contains configuration settings for machine-wide assembly binding, built-in remoting channels, and ASP.NET.

The configuration system first looks in the machine configuration file for the **<appSettings>** element and other configuration sections that a developer might define. It then looks in the application configuration file. To keep the machine configuration file manageable, it is best to put these settings in the application configuration file. However, putting the settings in the machine configuration file can make your system more maintainable. For example, if you have a third-party component that both your client and server application uses, it is easier to put the settings for that component in one place. In this case, the machine configuration file is the appropriate place for the settings, so you don't have the same settings in two different files.

> **Note**
>
> Deploying an application using XCOPY will not copy the settings in the machine configuration file.

For more information about how the common language runtime uses the machine configuration file for assembly binding, see How the Runtime Locates Assemblies.

Application configuration files

An application configuration file contains settings that are specific to an app. This file includes configuration settings that the common language runtime reads (such as assembly binding policy, remoting objects, and so on), and settings that the app can read.

The name and location of the application configuration file depend on the app's host, which can be one of the following:

- Executable–hosted app.

  These apps have two configuration files: a source configuration file, which is modified by the developer during development, and an output file that's distributed with the app.

  By default, the name of the source configuration file is *App.config*. When you create a .NET Framework project in Visual Studio, an *App.config* file is automatically added to the project. You can also add a file manually by selecting **File** > **New File**. Place the *App.config* file in the project directory and set its **Copy To Output Directory** property to **Copy always** or **Copy if newer**.

  To create the output configuration file that's deployed with the app, Visual Studio copies the source configuration file to the directory where the compiled assembly is placed. This file is named *<yourappname>.exe.config*. For example, an app named *myApp.exe* has an output configuration file named *myApp.exe.config*.

  In some cases, Visual Studio might modify the output configuration file. For more information, see Redirect versions at the app level.

- ASP.NET-hosted app.

  For more information about ASP.NET configuration files, see ASP.NET Configuration Settings.

Security configuration files

Security configuration files contain information about the code group hierarchy and permission sets associated with a policy level. We strongly recommend that you use the Code Access Security Policy tool (Caspol.exe) to modify security policy to ensure that policy changes do not corrupt the security configuration files.

> **Note**
>
> Starting with .NET Framework 4, the security configuration files are present only if security policy has been changed.

The security configuration files are in the following locations:

- Enterprise policy configuration file: %*runtime-install-path*%\Config\Enterprisesec.config
- Machine policy configuration file: %*runtime-install-path*%\Config\Security.config
- User policy configuration file: %USERPROFILE%\Application data\Microsoft\CLR security config\v*xx.xx*\Security.config

See also

- Configuration File Schema
- Specifying an Assembly's Location
- Redirecting Assembly Versions
- ASP.NET Web Site Administration
- Security Policy Management
- Caspol.exe (Code Access Security Policy Tool)
- Assemblies in .NET

How to: Read application settings

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 article shows you how to add a simple setting to an *App.config* file in a .NET Framework app, and then read the value programmatically. Instead of just reading a single value, you can read an entire section or the entire file. For more examples and information, see the ConfigurationManager docs.

Add the App.config file

Visual Studio makes it easy to add an *App.config* file to your project. After creating a .NET Framework project, right-click on your project in **Solution Explorer** and choose **Add** > **New Item**. Choose the **Application Configuration File** item and then select **Add**.

Add a setting

Open the *App.config* file and add the following XML within the `<configuration>` element.

XML

```xml
<appSettings>
  <add key="occupation" value="dentist"/>
</appSettings>
```

Access the setting programmatically

To access the setting's value in your code, get the value by indexing into the AppSettings property. The AppSettings property makes it easy to obtain data from the `<appSettings>` element of your configuration file.

C#

```
string occupation = ConfigurationManager.AppSettings["occupation"];
```

Configuration for libraries

While it's straightforward to use configuration files for executable apps, it's a little more complicated for class libraries. Class libraries can access configuration settings in the same way as executable apps, however, the configuration settings must exist in the client app's *App.config* file. Even if you distribute an *App.config* file alongside your library's assembly file, the library code will not read the file. Alternatively, consider the following ways to use configuration settings in a class library:

- Obtain the configuration settings in the client app and pass them to the class you're instantiating from the class library.
- Implement a custom section type that extends the ConfigurationSection class. Keep a separate configuration file for your class library, and then reference the library's configuration file from the client app's configuration file. For more information, see How to: Create Custom Configuration Sections Using ConfigurationSection.

See also

- System.Configuration.ConfigurationManager

How to: Locate Assemblies by Using DEVPATH

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.

Developers might want to make sure that a shared assembly they are building works correctly with multiple applications. Instead of continually putting the assembly in the global assembly cache during the development cycle, the developer can create a DEVPATH environment variable that points to the build output directory for the assembly.

For example, assume that you are building a shared assembly called MySharedAssembly and the output directory is C:\MySharedAssembly\Debug. You can put C:\MySharedAssembly\Debug in the DEVPATH variable. You must then specify the <developmentMode> element in the machine configuration file. This element tells the common language runtime to use DEVPATH to locate assemblies.

The shared assembly must be discoverable by the runtime. To specify a private directory for resolving assembly references use the <codeBase> Element or <probing> Element in a configuration file, as described in Specifying an Assembly's Location. You can also put the assembly in a subdirectory of the application directory. For more information, see How the Runtime Locates Assemblies.

> **Note**
>
> This is an advanced feature, intended only for development.

The following example shows how to cause the runtime to search for assemblies in directories specified by the DEVPATH environment variable.

Example

```
XML
```

```
<configuration>
  <runtime>
    <developmentMode developerInstallation="true"/>
  </runtime>
</configuration>
```

This setting defaults to false.

> **Note**
>
> Use this setting only at development time. The runtime does not check the versions on strong-named assemblies found in the DEVPATH. It simply uses the first assembly it finds.

See also

- Configuring Apps by using Configuration Files

Redirect assembly versions

Summarize this article for me

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

You can redirect compile-time binding references to .NET Framework assemblies, third-party assemblies, or your own app's assemblies. You can redirect your app to use a different version of an assembly in a number of ways: through publisher policy, through an app configuration file, or through the machine configuration file. This article discusses how assembly binding works in .NET Framework and how you can configure it.

> **Tip**
>
> This article is specific to .NET Framework apps. For information about assembly loading in .NET 5+ (and .NET Core), see Dependency loading in .NET.

Assembly unification and default binding

Bindings to .NET Framework assemblies are sometimes redirected through a process called *assembly unification*. .NET Framework consists of a version of the common language runtime and about two dozen .NET Framework assemblies that make up the type library. These .NET Framework assemblies are treated by the runtime as a single unit. By default, when an app is launched, all references to types in code run by the runtime are directed to .NET Framework assemblies that have the same version number as the runtime that's loaded in a process. The redirections that occur with this model are the default behavior for the runtime.

For example, if your app references types in the System.XML namespace and was built by using .NET Framework 4.5, it contains static references to the System.XML assembly that ships with runtime version 4.5. If you want to redirect the binding reference to point to the System.XML assembly that ships with .NET Framework 4, you can put redirect information in the app configuration file. A binding redirection in a configuration file for a unified .NET Framework assembly cancels the unification for that assembly.

In addition, you might want to manually redirect assembly binding for third-party assemblies if there are multiple versions available.

> **Tip**
>
> If you update a NuGet package that your application references indirectly and start to see new errors like `FileLoadException`, `MissingMethodException`, `TypeLoadException`, or `FileNotFoundException`, you might need to enable automatic binding redirects or manually add a binding redirect. This is normal when updating NuGet packages and is a result of some packages being built against an older version of a dependency. The following app config file excerpt adds a binding redirect for the System.Memory package:
>
> XML
>
> ```xml
> <dependentAssembly>
>    <assemblyIdentity name="System.Memory" publicKeyToken="cc7b13ffcd2ddd51"
> culture="neutral" />
>    <bindingRedirect oldVersion="0.0.0.0-4.0.2.0" newVersion="4.0.2.0" />
> </dependentAssembly>
> ```

Redirect versions by using publisher policy

Vendors of assemblies can direct apps to a newer version of an assembly by including a publisher policy file with the new assembly. The publisher policy file, which is located in the global assembly cache, contains assembly redirection settings.

Each *major.minor* version of an assembly has its own publisher policy file. For example, redirections from version 2.0.2.222 to 2.0.3.000 and from version 2.0.2.321 to version 2.0.3.000 both go into the same file, because they are associated with version 2.0. However, a redirection from version 3.0.0.999 to version 4.0.0.000 goes into the file for version 3.0.999. Each major version of the .NET Framework has its own publisher policy file.

If a publisher policy file exists for an assembly, the runtime checks this file after checking the assembly's manifest and app configuration file. Vendors should use publisher policy files only when the new assembly is backward compatible with the assembly being redirected.

You can bypass publisher policy for your app by specifying settings in the app configuration file, as discussed in the Bypass publisher policy section.

Redirect versions at the app level

There are a few different techniques for changing the binding behavior for your app through the app configuration file: you can manually edit the file, you can rely on automatic binding

redirection, or you can specify binding behavior by bypassing publisher policy.

Manually edit the app config file

You can manually edit the app configuration file to resolve assembly issues. For example, if a vendor releases a newer version of an assembly that your app uses without supplying a publisher policy (because they don't guarantee backward compatibility), you can direct your app to use the newer version of the assembly by putting assembly binding information in your app's configuration file as follows.

XML

```
<dependentAssembly>
  <assemblyIdentity name="someAssembly"
    publicKeyToken="32ab4ba45e0a69a1"
    culture="en-us" />
  <bindingRedirect oldVersion="7.0.0.0" newVersion="8.0.0.0" />
</dependentAssembly>
```

Rely on automatic binding redirection

When you create a desktop app in Visual Studio that targets .NET Framework 4.5.1 or a later version, the app uses automatic binding redirection. This means that if two components reference different versions of the same strong-named assembly, the runtime automatically adds a binding redirection to the newer version of the assembly in the output app configuration (app.config) file. This redirection overrides the assembly unification that might otherwise take place. The source app.config file is not modified. For example, let's say that your app directly references an out-of-band .NET Framework component but uses a third-party library that targets an older version of the same component. When you compile the app, the output app configuration file is modified to contain a binding redirection to the newer version of the component.

If you create a web app, you receive a build warning regarding the binding conflict, which in turn, gives you the option to add the necessary binding redirect to the source web configuration file.

If you manually add binding redirects to the source app.config file, at compile time, Visual Studio tries to unify the assemblies based on the binding redirects you added. For example, let's say you insert the following binding redirect for an assembly:

```
<bindingRedirect oldVersion="3.0.0.0" newVersion="2.0.0.0" />
```

If another project in your app references version 1.0.0.0 of the same assembly, automatic binding redirection adds the following entry to the output app.config file so that the app is unified on version 2.0.0.0 of this assembly:

```
<bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0" />
```

You can enable automatic binding redirection if your app targets older versions of .NET Framework. You can override this default behavior by providing binding redirection information in the app.config file for any assembly, or by turning off the binding redirection feature. For information about how to turn this feature on or off, see How to: Enable and Disable Automatic Binding Redirection.

Bypass publisher policy

You can override publisher policy in the app configuration file if necessary. For example, new versions of assemblies that claim to be backward compatible can still break an app. If you want to bypass publisher policy, add a <publisherPolicy> element to the <dependentAssembly> element in the app configuration file, and set the `apply` attribute to `no`, which overrides any previous `yes` settings.

```
<publisherPolicy apply="no" />
```

Bypass publisher policy to keep your app running for your users, but make sure you report the problem to the assembly vendor. If an assembly has a publisher policy file, the vendor should make sure that the assembly is backward compatible and that clients can use the new version as much as possible.

Redirect versions for tests, plugins, or libraries used by another component

For tests, you should generate a *.dll.config* file. Most existing unit test frameworks honor these files when loading tests.

Plugins might honor *.dll.config* files, however, they also might not. The only fool-proof mechanism for redirects is by providing `bindingRedirects` when the AppDomain is created.

You might try to solve this problem with AssemblyResolve event handlers, but that doesn't work since those handlers only are called on a failed load. If an assembly load succeeds, either because it was loaded by another assembly or the host, or was present in the GAC, an `AssemblyResolve` handler won't be called.

Generate binding redirects for unit test projects

Unit test projects compile to DLLs, not executables. Automatic binding redirect generation only applies to executable output types, so unit test projects don't receive binding redirects by default. When a unit test project references assemblies that have conflicting versions, tests might fail at runtime with exceptions such as FileLoadException.

To generate a *.dll.config* file with binding redirects for a unit test project, add both `AutoGenerateBindingRedirects` and `GenerateBindingRedirectsOutputType` to the project file:

XML

```xml
<PropertyGroup>
  <AutoGenerateBindingRedirects>true</AutoGenerateBindingRedirects>
  <GenerateBindingRedirectsOutputType>true</GenerateBindingRedirectsOutputType>
</PropertyGroup>
```

The `GenerateBindingRedirectsOutputType` property instructs MSBuild to generate a *.dll.config* file alongside the test assembly output, even though the output type is a class library (DLL). Most test runners, including VSTest, load this configuration file when running tests.

After adding these properties and rebuilding the project, verify that a *.dll.config* file appears in the build output directory next to the test assembly. The file contains the binding redirects needed to resolve assembly version conflicts at test run time.

Redirect versions at the machine level

There might be rare cases when a machine administrator wants all apps on a computer to use a specific version of an assembly. For example, a specific version might fix a security hole. If an assembly is redirected in the machine's configuration file, called *machine.config*, all apps on that machine that use the old version are directed to use the new version. The machine configuration file overrides the app configuration file and the publisher policy file. This *machine.config* file is located at *%windir%\Microsoft.NET\Framework[version]\config\machine.config* for 32-bit machines, or *%windir%\Microsoft.NET\Framework64[version]\config\machine.config* for 64-bit machines.

Specify assembly binding in configuration files

You use the same XML format to specify binding redirects whether it's in the app configuration file, the machine configuration file, or the publisher policy file. To redirect one assembly version to another, use the <bindingRedirect> element. The `oldVersion` attribute can specify a single assembly version or a range of versions. The `newVersion` attribute should specify a single

version. For example, `<bindingRedirect oldVersion="1.1.0.0-1.2.0.0" newVersion="2.0.0.0"/>` specifies that the runtime should use version 2.0.0.0 instead of the assembly versions between 1.1.0.0 and 1.2.0.0.

The following code example demonstrates a variety of binding redirect scenarios. The example specifies a redirect for a range of versions for `myAssembly`, and a single binding redirect for `mySecondAssembly`. The example also specifies that publisher policy file will not override the binding redirects for `myThirdAssembly`.

To bind an assembly, you must specify the string "urn:schemas-microsoft-com:asm.v1" with the `xmlns` attribute in the <assemblyBinding> tag.

XML

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="myAssembly"
          publicKeyToken="32ab4ba45e0a69a1"
          culture="en-us" />
        <!-- Assembly versions can be redirected in app,
          publisher policy, or machine configuration files. -->
        <bindingRedirect oldVersion="1.0.0.0-2.0.0.0" newVersion="3.0.0.0" />
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="mySecondAssembly"
          publicKeyToken="32ab4ba45e0a69a1"
          culture="en-us" />
            <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0" />
      </dependentAssembly>
      <dependentAssembly>
      <assemblyIdentity name="myThirdAssembly"
        publicKeyToken="32ab4ba45e0a69a1"
        culture="en-us" />
        <!-- Publisher policy can be set only in the app
          configuration file. -->
        <publisherPolicy apply="no" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Limit assembly bindings to a specific version

You can use the `appliesTo` attribute on the <assemblyBinding> element in an app configuration file to redirect assembly binding references to a specific version of .NET Framework. This optional attribute uses a .NET Framework version number to indicate what

version it applies to. If no `appliesTo` attribute is specified, the <assemblyBinding> element applies to all versions of .NET Framework.

For example, to redirect assembly binding for a .NET Framework 3.5 assembly, you'd include the following XML code in your app configuration file.

XML

```xml
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1"
    appliesTo="v3.5">
    <dependentAssembly>
      <!-- assembly information goes here -->
    </dependentAssembly>
  </assemblyBinding>
</runtime>
```

You should enter redirection information in version order. For example, enter assembly binding redirection information for .NET Framework 3.5 assemblies followed by .NET Framework 4.5 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 .NET Framework. If there is 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 3.5 assembly and another reference to a .NET Framework 4 assembly, use the pattern shown in the following pseudocode.

XML

```xml
<assemblyBinding xmlns="..." appliesTo="v3.5 ">
  <!--.NET Framework version 3.5 redirects here -->
</assemblyBinding>

<assemblyBinding xmlns="..." appliesTo="v4.0.30319">
  <!--.NET Framework version 4.0 redirects here -->
</assemblyBinding>

<assemblyBinding xmlns="...">
  <!-- redirects meant for all versions of the runtime -->
</assemblyBinding>
```

See also

- How to: Enable and Disable Automatic Binding Redirection
- <bindingRedirect> Element

- Assembly Binding Redirection Security Permission
- Assemblies in .NET
- Programming with Assemblies
- How the Runtime Locates Assemblies
- Configuring Apps
- Runtime Settings Schema
- Configuration File Schema
- How to: Create a Publisher Policy

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 03/03/2026

How to: Enable and disable automatic binding redirection

Article • 07/29/2022

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

When you compile desktop apps in Visual Studio that target .NET Framework 4.5.1 and later versions, binding redirects may be automatically added to the app configuration file to override assembly unification. Binding redirects are added if your app or its components reference more than one version of the same assembly, even if you manually specify binding redirects in the configuration file for your app. The automatic binding redirection feature affects desktop apps that target .NET Framework 4.5.1 or a later version. If you haven't explicitly enabled or disabled autogenerated binding redirection and you upgrade an existing project, the feature is automatically enabled.

For web apps, when Visual Studio encounters a binding conflict, it prompts you to add a binding redirect to resolve the conflict.

You can *enable* automatic binding redirection for existing apps that target previous versions of .NET Framework (4.5 and earlier). You can *disable* this feature if you want to manually author binding redirects.

> **Important**
>
> Starting with Visual Studio 2022, Visual Studio no longer includes .NET Framework components for .NET Framework 4.0 - 4.5.1 because these versions are no longer supported. Visual Studio 2022 and later versions can't build apps that target .NET Framework 4.0 through .NET Framework 4.5.1. To continue building these apps, you can use Visual Studio 2019 or an earlier version.

Disable automatic binding redirects in desktop apps

Automatic binding redirects are enabled by default for Windows desktop apps that target .NET Framework 4.5.1 and later versions. The binding redirects are added to the

output configuration (**app.config**) file when the app is compiled. The redirects override the assembly unification that might otherwise take place. The source **app.config** file is not modified. You can disable this feature by modifying the project file for the app or by deselecting a checkbox in the project's properties in Visual Studio.

Disable through project properties

If you have Visual Studio 2017 version 15.7 or later, you can disable autogenerated binding redirects in the project's property pages.

1. Right-click the project in **Solution Explorer** and select **Properties**.
2. On the **Application** page, uncheck the **Auto-generate binding redirects** option.

   If you don't see the option, you'll need to manually disable the feature in the project file.
3. Press `Ctrl` + `S` to save the change.

Disable manually in the project file

1. Open the project file for editing using one of the following methods:

   - In Visual Studio, select the project in **Solution Explorer**, and then choose **Open Folder in File Explorer** from the shortcut menu. In File Explorer, find the project (.csproj or .vbproj) file and open it in Notepad.
   - In Visual Studio, in **Solution Explorer**, right-click the project and choose **Unload Project**. Right-click the unloaded project again, and then choose **Edit [projectname.csproj]**.
2. In the project file, find the following property entry:

   ```
   XML

   <AutoGenerateBindingRedirects>true</AutoGenerateBindingRedirects>
   ```
3. Change `true` to `false`:

   ```
   XML

   <AutoGenerateBindingRedirects>false</AutoGenerateBindingRedirects>
   ```

Enable automatic binding redirects manually

You can enable automatic binding redirects in existing apps that target older versions of .NET Framework, or in cases where you're not automatically prompted to add a redirect. If you're targeting a newer version of .NET Framework but do not get automatically prompted to add a redirect, you'll likely get build output that suggests you remap assemblies.

1. Open the project file for editing using one of the following methods:

   - In Visual Studio, select the project in **Solution Explorer**, and then choose **Open Folder in File Explorer** from the shortcut menu. In File Explorer, find the project (.csproj or .vbproj) file and open it in Notepad.
   - In Visual Studio, in **Solution Explorer**, right-click the project and choose **Unload Project**. Right-click the unloaded project again, and then choose **Edit [projectname.csproj]**.

2. Add the following element to the first configuration property group (under the <PropertyGroup> tag):

   XML

   ```
   <AutoGenerateBindingRedirects>true</AutoGenerateBindingRedirects>
   ```

   The following shows an example project file with the element inserted:

   XML

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <Project ToolsVersion="12.0" DefaultTargets="Build"
   xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
     <Import
   Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Comm
   on.props"
   Condition="Exists('$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Micr
   osoft.Common.props')" />
     <PropertyGroup>
       <Configuration Condition=" '$(Configuration)' == ''
   ">Debug</Configuration>
       <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
       <ProjectGuid>{123334}</ProjectGuid>
       ...
       <AutoGenerateBindingRedirects>true</AutoGenerateBindingRedirects>
     </PropertyGroup>
     ...
   </Project>
   ```

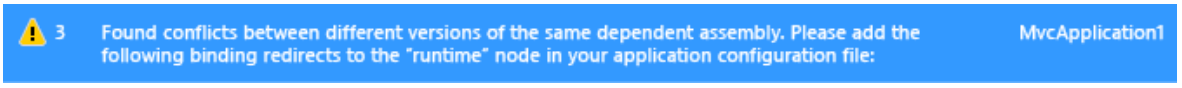
3. Compile your app.

Enable automatic binding redirects in web apps

Automatic binding redirects are implemented differently for web apps. Because the source configuration (**web.config**) file must be modified for web apps, binding redirects are not automatically added to the configuration file. However, Visual Studio notifies you of binding conflicts, and you can add binding redirects to resolve the conflicts. Because you're always prompted to add binding redirects, you don't need to explicitly disable this feature for a web app.

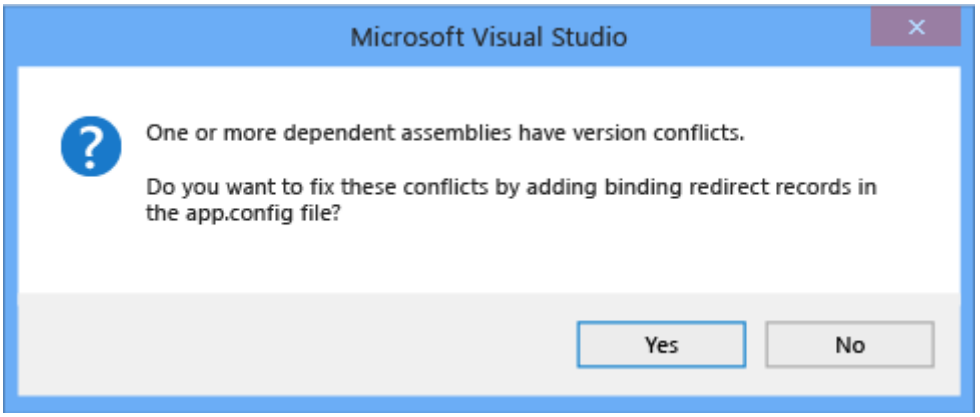
To add binding redirects to a **web.config** file:

1. In Visual Studio, compile the app, and check for build warnings.

2. If there are assembly binding conflicts, a warning appears. Double-click the warning, or select the warning and press `Enter`.

   A dialog box that enables you to automatically add the necessary binding redirects to the source **web.config** file appears.

See also

- <bindingRedirect> Element
- Redirecting Assembly Versions

Assembly Binding Redirection Security Permission

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.

Explicit assembly binding redirection in an application configuration file requires a security permission. This applies to redirection of .NET Framework assemblies and assemblies from third parties. The permission is granted by setting the SecurityPermissionFlag flag on the SecurityPermission. Managed assemblies have no permissions by default.

The security permission is granted to applications running in the Trusted Zone (local machine) and Intranet Zone. Applications running in the Internet Zone are strictly prohibited from performing assembly binding redirection.

The permission is not required if assembly redirection is performed in a publisher policy file that is controlled by the component publisher, or in the machine configuration file that is controlled by the administrator. However, the permission is required for an application to explicitly ignore publisher policy using the <publisherPolicy apply="no"/> element in the application configuration file.

The following table shows the default security settings for the **BindingRedirects** flag.

Expand table

Zone	BindingRedirects flag setting
Trusted Zone (local machine)	**ON**
Intranet Zone	**ON**
Internet Zone	**OFF**
Untrusted zones	**OFF**

An administrator can change these security settings to support or restrict specific scenarios on a given computer. There are no tools for changing the **BindingRedirects**

flag setting from the default; an administrator must manually edit the Security.config file on a user's computer.

See also

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

Specifying an Assembly's Location

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

There are two ways to specify an assembly's location:

- Using the <codeBase> element.
- Using the <probing> element.

You can also use the .NET Framework Configuration Tool (Mscorcfg.msc) to specify assembly locations or specify locations for the common language runtime to probe for assemblies.

Using the `<codeBase>` Element

You can use the `<codeBase>` element only in machine configuration or publisher policy files that also redirect the assembly version. When the runtime determines which assembly version to use, it applies the code base setting from the file that determines the version. If no code base is indicated, the runtime probes for the assembly in the normal way. For details, see How the Runtime Locates Assemblies.

The following example shows how to specify an assembly's location.

XML

```xml
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
       <dependentAssembly>
         <assemblyIdentity name="myAssembly"
                           publicKeyToken="32ab4ba45e0a69a1"
                           culture="en-us" />
         <codeBase version="2.0.0.0"
                   href="http://www.litwareinc.com/myAssembly.dll"/>
       </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

The `version` attribute is required for all strong-named assemblies but should be omitted for assemblies that aren't strong-named. The `<codeBase>` element requires the `href` attribute. You cannot specify version ranges in the `<codeBase>` element.

> **Note**
>
> If you are supplying a code base hint for an assembly that is not strong-named, the hint must point to the application base or a subdirectory of the application base directory.

Using the `<probing>` Element

The runtime locates assemblies that do not have a code base by probing. For more information about probing, see How the Runtime Locates Assemblies.

You can use the <probing> element in the application configuration file to specify subdirectories the runtime should search when locating an assembly. The following example shows how to specify directories the runtime should search.

XML

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <probing privatePath="bin;bin2\subbin;bin3"/>
      </assemblyBinding>
   </runtime>
</configuration>
```

The `privatePath` attribute contains the directories that the runtime should search for assemblies. If the application is located at C:\Program Files\MyApp, the runtime will look for assemblies that do not specify a code base in C:\Program Files\MyApp\Bin, C:\Program Files\MyApp\Bin2\Subbin, and C:\Program Files\MyApp\Bin3. The directories specified in `privatePath` must be subdirectories of the application base directory.

See also

- Assemblies in .NET
- Programming with Assemblies
- How the Runtime Locates Assemblies
- Configuring Apps by using Configuration Files

Last updated on 12/02/2025

Configuring Cryptography Classes

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 Windows SDK allows computer administrators to configure the default cryptographic algorithms and algorithm implementations that the .NET Framework and appropriately written applications use. For example, an enterprise that has its own implementation of a cryptographic algorithm can make that implementation the default instead of the implementation shipped in the Windows SDK. Although managed applications that use cryptography can always choose to explicitly bind to a particular implementation, it is recommended that they create cryptographic objects by using the crypto configuration system.

In This Section

Mapping Algorithm Names to Cryptography Classes Describes how to map an algorithm name to a cryptography class.

Mapping Object Identifiers to Cryptography Algorithms Describes how to map an object identifier to a cryptography algorithm.

Related Sections

Cryptographic Services Provides an overview of cryptographic services provided by the Windows SDK.

Cryptography Settings Schema Describes elements that map friendly algorithm names to classes that implement cryptography algorithms.

Mapping Algorithm Names to Cryptography Classes

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.

There are four ways a developer can create a cryptography object using the Windows SDK:

- Create an object by using the **new** operator.
- Create an object that implements a particular cryptography algorithm by calling the **Create** method on the abstract class for that algorithm.
- Create an object that implements a particular cryptography algorithm by calling the CryptoConfig.CreateFromName method.
- Create an object that implements a class of cryptographic algorithms (such as a symmetric block cipher) by calling the **Create** method on the abstract class for that type of algorithm (such as SymmetricAlgorithm).

For example, suppose a developer wants to compute the SHA1 hash of a set of bytes. The System.Security.Cryptography namespace contains two implementations of the SHA1 algorithm, one purely managed implementation and one that wraps CryptoAPI. The developer can choose to instantiate a particular SHA1 implementation (such as the SHA1Managed) by calling the **new** operator. However, if it does not matter which class the common language runtime loads as long as the class implements the SHA1 hash algorithm, the developer can create an object by calling the SHA1.Create method. This method calls **System.Security.Cryptography.CryptoConfig.CreateFromName("System.Security.Cryptography.SHA1")**, which must return an implementation of the SHA1 hash algorithm.

The developer can also call **System.Security.Cryptography.CryptoConfig.CreateFromName("SHA1")** because, by default, cryptography configuration includes short names for the algorithms shipped in the .NET Framework.

If it does not matter which hash algorithm is used, the developer can call the HashAlgorithm.Create method, which returns an object that implements a hashing transformation.

Mapping Algorithm Names in Configuration Files

By default, the runtime returns a SHA1CryptoServiceProvider object for all four scenarios. However, a machine administrator can change the type of object that the methods in the last two scenarios return. To do this, you must map a friendly algorithm name to the class you want to use in the machine configuration file (Machine.config).

The following example shows how to configure the runtime so that **System.Security.Cryptography.SHA1.Create**, **System.Security.CryptoConfig.CreateFromName("SHA1")**, and **System.Security.Cryptography.HashAlgorithm.Create** return a `MySHA1HashClass` object.

XML

```
<configuration>
   <!-- Other configuration settings. -->
   <mscorlib>
      <cryptographySettings>
         <cryptoNameMapping>
            <cryptoClasses>
               <cryptoClass MySHA1Hash="MySHA1HashClass, MyAssembly
                  Culture='en', PublicKeyToken=a5d015c7d5a0b012,
                  Version=1.0.0.0"/>
            </cryptoClasses>
            <nameEntry name="SHA1" class="MySHA1Hash"/>
            <nameEntry name="System.Security.Cryptography.SHA1"
                       class="MySHA1Hash"/>
            <nameEntry name="System.Security.Cryptography.HashAlgorithm"
                       class="MySHA1Hash"/>
         </cryptoNameMapping>
      </cryptographySettings>
   </mscorlib>
</configuration>
```

You can specify the name of the attribute in the <cryptoClass> element (the previous example names the attribute `MySHA1Hash`). The value of the attribute in the **<cryptoClass>** element is a string that the common language runtime uses to find the class. You can use any string that meets the requirements specified in Specifying Fully Qualified Type Names.

Many algorithm names can map to the same class. The <nameEntry> element maps a class to one friendly algorithm name. The **name** attribute can be either a string that is used when calling the **System.Security.Cryptography.CryptoConfig.CreateFromName** method or the name of an abstract cryptography class in the System.Security.Cryptography namespace. The value of the **class** attribute is the name of the attribute in the **<cryptoClass>** element.

> **Note**
>
> You can get an SHA1 algorithm by calling the SHA1.Create or the **Security.CryptoConfig.CreateFromName("SHA1")** method. Each method guarantees only that it returns an object that implements the SHA1 algorithm. You do not have to map each friendly name of an algorithm to the same class in the configuration file.

For a list of default names and the classes they map to, see CryptoConfig.

See also

- Cryptographic Services
- Configuring Cryptography Classes

Mapping Object Identifiers to Cryptography Algorithms

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.

Digital signatures ensure that data is not tampered with when it is sent from one program to another. Typically the digital signature is computed by applying a mathematical function to the hash of the data to be signed. When formatting a hash value to be signed, some digital signature algorithms append an ASN.1 Object Identifier (OID) as part of the formatting operation. The OID identifies the algorithm that was used to compute the hash. You can map algorithms to object identifiers to extend the cryptography mechanism to use custom algorithms. The following example shows how to map an object identifier to a new hash algorithm.

XML

```
<configuration>
   <mscorlib>
      <cryptographySettings>
         <cryptoNameMapping>
            <cryptoClasses>
               <cryptoClass MyNewHash="MyNewHashClass, MyAssembly
                  Culture='en', PublicKeyToken=a5d015c7d5a0b012,
                  Version=1.0.0.0"/>
            </cryptoClasses>
            <nameEntry name="NewHash" class="MyNewHash"/>
         </cryptoNameMapping>
         <oidMap>
            <oidEntry OID="1.3.14.33.42.46"  name="NewHash"/>
         </oidMap>
      </cryptographySettings>
   </mscorlib>
</configuration>
```

The <oidEntry> element contains two attributes. The **OID** attribute is the object identifier number. The **name** attribute is the value of the **name** attribute from the <nameEntry> element. There must be a mapping from an algorithm name to a class before an object identifier can be mapped to a simple name.

See also

- Configuring Cryptography Classes
- Cryptographic Services

How to: Create a Publisher Policy

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.

Vendors of assemblies can state that applications should use a newer version of an assembly by including a publisher policy file with the upgraded assembly. The publisher policy file specifies assembly redirection and code base settings, and uses the same format as an application configuration file. The publisher policy file is compiled into an assembly and placed in the global assembly cache.

There are three steps involved in creating a publisher policy:

1. Create a publisher policy file.
2. Create a publisher policy assembly.
3. Add the publisher policy assembly to the global assembly cache.

The schema for publisher policy is described in Redirecting Assembly Versions. The following example shows a publisher policy file that redirects one version of `myAssembly` to another.

XML

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
       <dependentAssembly>
         <assemblyIdentity name="myAssembly"
                           publicKeyToken="32ab4ba45e0a69a1"
                           culture="en-us" />
         <!-- Redirecting to version 2.0.0.0 of the assembly. -->
         <bindingRedirect oldVersion="1.0.0.0"
                          newVersion="2.0.0.0"/>
       </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

To learn how to specify a code base, see Specifying an Assembly's Location.

Creating the Publisher Policy Assembly

Use the Assembly Linker (Al.exe) to create the publisher policy assembly.

To create a publisher policy assembly

Type the following command at the command prompt:

```
Console

al /link:publisherPolicyFile /out:publisherPolicyAssemblyFile
/keyfile:keyPairFile /platform:processorArchitecture
```

In this command:

- The `publisherPolicyFile` argument is the name of the publisher policy file.
- The `publisherPolicyAssemblyFile` argument is the name of the publisher policy assembly that results from this command. The assembly file name must follow the format:

  `policy.majorNumber.minorNumber.mainAssemblyName.dll'
- The `keyPairFile` argument is the name of the file containing the key pair. You must sign the assembly and publisher policy assembly with the same key pair.
- The `processorArchitecture` argument identifies the platform targeted by a processor-specific assembly.

  > **Note**
  >
  > The ability to target a specific processor architecture is available starting with .NET Framework 2.0.

The ability to target a specific processor architecture is available starting with .NET Framework 2.0. The following command creates a publisher policy assembly called `policy.1.0.myAssembly` from a publisher policy file called `pub.config`, assigns a strong name to the assembly using the key pair in the `sgKey.snk` file, and specifies that the assembly targets the x86 processor architecture.

```
Console
```

```
al /link:pub.config /out:policy.1.0.myAssembly.dll /keyfile:sgKey.snk
/platform:x86
```

The publisher policy assembly must match the processor architecture of the assembly that it applies to. Thus, if your assembly has a ProcessorArchitecture value of MSIL, the publisher policy assembly for that assembly must be created with `/platform:anycpu`. You must provide a separate publisher policy assembly for each processor-specific assembly.

A consequence of this rule is that in order to change the processor architecture for an assembly, you must change the major or minor component of the version number, so that you can supply a new publisher policy assembly with the correct processor architecture. The old publisher policy assembly cannot service your assembly once your assembly has a different processor architecture.

Another consequence is that the version 2.0 linker cannot be used to create a publisher policy assembly for an assembly compiled using earlier versions of the .NET Framework, because it always specifies processor architecture.

Adding the Publisher Policy Assembly to the Global Assembly Cache

Use the Global Assembly Cache tool (Gacutil.exe) to add the publisher policy assembly to the global assembly cache.

To add the publisher policy assembly to the global assembly cache

Type the following command at the command prompt:

Console

```
gacutil /i publisherPolicyAssemblyFile
```

The following command adds `policy.1.0.myAssembly.dll` to the global assembly cache.

Console

```
gacutil /i policy.1.0.myAssembly.dll
```

Important

The publisher policy assembly cannot be added to the global assembly cache unless the original publisher policy file specified in the `/link` argument is located in the same directory as the assembly.

See also

- Programming with Assemblies
- How the Runtime Locates Assemblies
- Configuring Apps by using Configuration Files
- Runtime Settings Schema
- Configuration File Schema
- Redirecting Assembly Versions

Configuration file schema for .NET Framework

Article • 05/25/2022

Configuration files are standard XML files that you can use to change settings and set policies for your apps. The .NET Framework configuration schema consists of elements that you can use in configuration files to control the behavior of your apps. The table of contents for this section reflects the schema hierarchy for startup, runtime, network, and other types of configuration settings.

For information about the types, format, and location of configuration files, see Configure apps. Familiarize yourself with XML if you want to edit the configuration files directly.

> ⓘ **Important**
>
> XML tags and attributes in configuration files are case-sensitive.

In this section

<configuration> Element
The top-level element for all configuration files.

<assemblyBinding> Element
Specifies assembly binding policy at the configuration level.

<linkedConfiguration> Element
Specifies a configuration file to include.

Startup Settings Schema
Elements that specify which version of the common language runtime to use.

Runtime Settings Schema
Elements that configure assembly binding and runtime behavior.

Network Settings Schema
Elements that specify how the .NET Framework connects to the internet.

Cryptography Settings Schema
Elements that map friendly algorithm names to classes that implement cryptography algorithms.

Configuration Sections Schema
Elements used to create and use configuration sections for custom settings.

Trace and Debug Settings Schema
Elements that specify trace switches and listeners.

Compiler and Language Provider Settings Schema
Elements that specify compiler configuration for available language providers.

Application Settings Schema
Elements that enable a Windows Forms or ASP.NET application to store and retrieve application-scoped and user-scoped settings.

App Settings Schema
Contains custom application settings, such as file paths, XML Web service URLs, or any other custom configuration information for an application.

Web Settings Schema
Elements for configuring how ASP.NET works with a host application such as IIS. Used in *Aspnet.config* files.

Windows Forms Configuration Schema
All elements in the Windows Forms application configuration section, which includes customizations such as multi-monitor and high-DPI support.

WCF Configuration Schema
All elements that enable you to configure WCF service and client applications.

WCF Directive Syntax
Describes the `@ServiceHost` directive, which defines page-specific attributes used by the .svc compiler.

Related sections

Remoting Settings Schema
Describes the elements that configure client and server applications that implement remoting.

ASP.NET Settings Schema
Describes the elements that control the behavior of ASP.NET Web applications.

Web Services Settings Schema
Describes the elements that control the behavior of ASP.NET Web services and their clients.

Configuring .NET Framework Apps

Describes how to configure security, assembly binding, and remoting in the .NET Framework.