Select the .NET version to use

Article • 02/28/2023

This article explains the policies used by the .NET tools, SDK, and runtime for selecting versions. These policies provide a balance between running applications using the specified versions and enabling ease of upgrading both developer and end-user machines. These policies enable:

- Easy and efficient deployment of .NET, including security and reliability updates.
- Use the latest tools and commands independent of target runtime.

Version selection occurs:

- When you run an SDK command, the SDK uses the latest installed version.
- When you build an assembly, target framework monikers define build time APIs.
- When you run a .NET application, target framework dependent apps roll-forward.
- When you publish a self-contained application, self-contained deployments include the selected runtime.

The rest of this document examines those four scenarios.

The SDK uses the latest installed version

SDK commands include `dotnet new` and `dotnet run`. The .NET CLI must choose an SDK version for every `dotnet` command. It uses the latest SDK installed on the machine by default, even if:

- The project targets an earlier version of the .NET runtime.
- The latest version of the .NET SDK is a preview version.

You can take advantage of the latest SDK features and improvements while targeting earlier .NET runtime versions. You can target different runtime versions of .NET using the same SDK tools.

On rare occasions, you may need to use an earlier version of the SDK. You specify that version in a *global.json* file. The "use latest" policy means you only use *global.json* to specify a .NET SDK version earlier than the latest installed version.

global.json can be placed anywhere in the file hierarchy. The CLI searches upward from the project directory for the first *global.json* it finds. You control which projects a given *global.json* applies to by its place in the file system. The .NET CLI searches for a *global.json* file iteratively navigating the path upward from the current working

directory. The first *global.json* file found specifies the version used. If that SDK version is installed, that version is used. If the SDK specified in the *global.json* isn't found, the .NET CLI uses matching rules to select a compatible SDK, or fails if none is found.

The following example shows the *global.json* syntax:

JSON

```
{
  "sdk": {
    "version": "5.0.0"
  }
}
```

The process for selecting an SDK version is:

1. `dotnet` searches for a *global.json* file iteratively reverse-navigating the path upward from the current working directory.
2. `dotnet` uses the SDK specified in the first *global.json* found.
3. `dotnet` uses the latest installed SDK if no *global.json* is found.

For more information about SDK version selection, see the Matching rules and rollForward sections of the global.json overview article.

Target framework monikers define build time APIs

You build your project against APIs defined in a **target framework moniker** (TFM). You specify the target framework in the project file. Set the `TargetFramework` element in your project file as shown in the following example:

XML

```
<TargetFramework>net8.0</TargetFramework>
```

You can build your project against multiple TFMs. Setting multiple target frameworks is more common for libraries but can be done with applications as well. You specify a `TargetFrameworks` property (plural of `TargetFramework`). The target frameworks are semicolon-delimited as shown in the following example:

XML

```
<TargetFrameworks>net8.0;net47</TargetFrameworks>
```

A given SDK supports a fixed set of frameworks, capped to the target framework of the runtime it ships with. For example, the .NET 8 SDK includes the .NET 8 runtime, which is an implementation of the `net8.0` target framework. The .NET 8 SDK supports `net7.0`, `net6.0`, and `net5.0`, but not `net9.0` (or higher). You install the .NET 9 SDK to build for `net9.0`.

.NET Standard

.NET Standard was a way to target an API surface shared by different implementations of .NET. Starting with the release of .NET 5, which is an API standard itself, .NET Standard has little relevance, except for one scenario: .NET Standard is useful when you want to target both .NET and .NET Framework. .NET 5 implements all .NET Standard versions.

For more information, see .NET 5 and .NET Standard.

Framework-dependent apps roll-forward

When you run an application from source with dotnet run, from a **framework-dependent deployment** with dotnet myapp.dll, or from a **framework-dependent executable** with `myapp.exe`, the `dotnet` executable is the **host** for the application.

The host chooses the latest patch version installed on the machine. For example, if you specified `net5.0` in your project file, and `5.0.2` is the latest .NET runtime installed, the `5.0.2` runtime is used.

If no acceptable `5.0.*` version is found, a new `5.*` version is used. For example, if you specified `net5.0` and only `5.1.0` is installed, the application runs using the `5.1.0` runtime. This behavior is referred to as "minor version roll-forward." Lower versions also won't be considered. When no acceptable runtime is installed, the application won't run.

A few usage examples demonstrate the behavior, if you target 5.0:

- ✓ 5.0 is specified. 5.0.3 is the highest patch version installed. 5.0.3 is used.
- ✗ 5.0 is specified. No 5.0.* versions are installed. 3.1.1 is the highest runtime installed. An error message is displayed.
- ✓ 5.0 is specified. No 5.0.* versions are installed. 5.1.0 is the highest runtime version installed. 5.1.0 is used.
- ✗ 3.0 is specified. No 3.x versions are installed. 5.0.0 is the highest runtime installed. An error message is displayed.

Minor version roll-forward has one side-effect that may affect end users. Consider the following scenario:

1. The application specifies that 5.0 is required.
2. When run, version 5.0.* isn't installed, however, 5.1.0 is. Version 5.1.0 will be used.
3. Later, the user installs 5.0.3 and runs the application again, 5.0.3 will now be used.

It's possible that 5.0.3 and 5.1.0 behave differently, particularly for scenarios like serializing binary data.

Control roll-forward behavior

> Before overriding default roll-forward behavior, familiarize yourself with the level of .NET runtime compatibility.

The roll-forward behavior for an application can be configured in four different ways:

1. Project-level setting by setting the `<RollForward>` property:

   XML

   ```xml
   <PropertyGroup>
     <RollForward>LatestMinor</RollForward>
   </PropertyGroup>
   ```

2. The `*.runtimeconfig.json` file.

   This file is produced when you compile your application. If the `<RollForward>` property was set in the project, it's reproduced in the `*.runtimeconfig.json` file as the `rollForward` setting. Users can edit this file to change the behavior of your application.

   JSON

   ```json
   {
     "runtimeOptions": {
       "tfm": "net5.0",
       "rollForward": "LatestMinor",
       "framework": {
         "name": "Microsoft.NETCore.App",
         "version": "5.0.0"
       }
     }
   }
   ```

3. The `dotnet` command's `--roll-forward <value>` property.

   When you run an application, you can control the roll-forward behavior through the command line:

   .NET CLI

   ```
   dotnet run --roll-forward LatestMinor
   dotnet myapp.dll --roll-forward LatestMinor
   myapp.exe --roll-forward LatestMinor
   ```

4. The `DOTNET_ROLL_FORWARD` environment variable.

Precedence

Roll forward behavior is set by the following order when your app is run, higher numbered items taking precedence over lower numbered items:

1. First the `*.runtimeconfig.json` config file is evaluated.
2. Next, the `DOTNET_ROLL_FORWARD` environment variable is considered, overriding the previous check.
3. Finally, any `--roll-forward` parameter passed to the running application overrides everything else.

Values

However you set the roll-forward setting, use one of the following values to set the behavior:

Expand table

Value	Description
`Minor`	**Default** if not specified. Roll-forward to the lowest higher minor version, if requested minor version is missing. If the requested minor version is present, then the `LatestPatch` policy is used.
`Major`	Roll-forward to the next available higher major version, and lowest minor version, if requested major version is missing. If the requested major version is present, then the `Minor` policy is used.
`LatestPatch`	Roll-forward to the highest patch version. This value disables minor version roll-forward.

Value	Description
`LatestMinor`	Roll-forward to highest minor version, even if requested minor version is present.
`LatestMajor`	Roll-forward to highest major and highest minor version, even if requested major is present.
`Disable`	Don't roll-forward, only bind to the specified version. This policy isn't recommended for general use since it disables the ability to roll-forward to the latest patches. This value is only recommended for testing.

Self-contained deployments include the selected runtime

You can publish an application as a **self-contained distribution**. This approach bundles the .NET runtime and libraries with your application. Self-contained deployments don't have a dependency on runtime environments. Runtime version selection occurs at publishing time, not run time.

The *restore* event that occurs when publishing selects the latest patch version of the given runtime family. For example, `dotnet publish` will select .NET 5.0.3 if it's the latest patch version in the .NET 5 runtime family. The target framework (including the latest installed security patches) is packaged with the application.

An error occurs if the minimum version specified for an application isn't satisfied. `dotnet publish` binds to the latest runtime patch version (within a given major.minor version family). `dotnet publish` doesn't support the roll-forward semantics of `dotnet run`. For more information about patches and self-contained deployments, see the article on runtime patch selection in deploying .NET applications.

Self-contained deployments may require a specific patch version. You can override the minimum runtime patch version (to higher or lower versions) in the project file, as shown in the following example:

XML

```xml
<PropertyGroup>
  <RuntimeFrameworkVersion>5.0.7</RuntimeFrameworkVersion>
</PropertyGroup>
```

The `RuntimeFrameworkVersion` element overrides the default version policy. For self-contained deployments, the `RuntimeFrameworkVersion` specifies the *exact* runtime

framework version. For framework-dependent applications, the `RuntimeFrameworkVersion` specifies the *minimum* required runtime framework version.

See also

- Download and install .NET.
- How to remove the .NET Runtime and SDK.