.NET project SDKs

Article • 01/06/2023 • 9 minutes to read

.NET Core and .NET 5 and later projects are associated with a software development kit (SDK). Each *project SDK* is a set of MSBuild targets and associated tasks that are responsible for compiling, packing, and publishing code. A project that references a project SDK is sometimes referred to as an *SDK-style project*.

Available SDKs

The following SDKs are available:

ID	Description	Repo
`Microsoft.NET.Sdk`	The .NET SDK	https://github.com/dotnet/sdk
`Microsoft.NET.Sdk.Web`	The .NET Web SDK	https://github.com/dotnet/sdk
`Microsoft.NET.Sdk.BlazorWebAssembly`	The .NET Blazor WebAssembly SDK	
`Microsoft.NET.Sdk.Razor`	The .NET Razor SDK	
`Microsoft.NET.Sdk.Worker`	The .NET Worker Service SDK	
`Microsoft.NET.Sdk.WindowsDesktop`	The .NET Desktop SDK, which includes Windows Forms (WinForms) and Windows Presentation Foundation (WPF).*	https://github.com/dotnet/winforms and https://github.com/dotnet/wpf

The .NET SDK is the base SDK for .NET. The other SDKs reference the .NET SDK, and projects that are associated with the other SDKs have all the .NET SDK properties available to them. The Web SDK, for example, depends on both the .NET SDK and the Razor SDK.

You can also author your own SDK that can be distributed via NuGet.

* Starting in .NET 5, Windows Forms and Windows Presentation Foundation (WPF) projects should specify the .NET SDK (`Microsoft.NET.Sdk`) instead of `Microsoft.NET.Sdk.WindowsDesktop`. For these projects, setting `TargetFramework` to `net5.0-windows` and `UseWPF` or `UseWindowsForms` to `true` will automatically import the Windows desktop SDK. If your project targets .NET 5 or later and specifies the `Microsoft.NET.Sdk.WindowsDesktop` SDK, you'll get build warning NETSDK1137.

Project files

.NET projects are based on the MSBuild format. Project files, which have extensions like *.csproj* for C# projects and *.fsproj* for F# projects, are in XML format. The root element of an MSBuild project file is the Project element. The `Project` element has an optional `Sdk` attribute that specifies which SDK (and version) to use. To use the .NET tools and build your code, set the `Sdk` attribute to one of the IDs in the Available SDKs table.

XML

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

To specify an SDK that comes from NuGet, include the version at the end of the name, or specify the name and version in the *global.json* file.

XML

```xml
<Project Sdk="MSBuild.Sdk.Extras/2.0.54">
  ...
</Project>
```

Another way to specify the SDK is with the top-level Sdk element:

XML

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

Referencing an SDK in one of these ways greatly simplifies project files for .NET. While evaluating the project, MSBuild adds implicit imports for `Sdk.props` at the top of the project file and `Sdk.targets` at the bottom.

XML

```xml
<Project>
  <!-- Implicit top import -->
  <Import Project="Sdk.props" Sdk="Microsoft.NET.Sdk" />
  ...
  <!-- Implicit bottom import -->
  <Import Project="Sdk.targets" Sdk="Microsoft.NET.Sdk" />
</Project>
```

Tip

On a Windows machine, the *Sdk.props* and *Sdk.targets* files can be found in the *%ProgramFiles%\dotnet\sdk\[version]\Sdks\Microsoft.NET.Sdk\Sdk* folder.

Preprocess the project file

You can see the fully expanded project as MSBuild sees it after the SDK and its targets are included by using the `dotnet msbuild -preprocess` command. The preprocess switch of the dotnet msbuild command shows which files are imported, their sources, and their contributions to the build without actually building the project.

If the project has multiple target frameworks, focus the results of the command on only one framework by specifying it as an MSBuild property. For example:

```
dotnet msbuild -property:TargetFramework=netcoreapp2.0 -preprocess:output.xml
```

Default includes and excludes

The default includes and excludes for Compile items, embedded resources, and None items are defined in the SDK. Unlike non-SDK .NET Framework projects, you don't need to specify these items in your project file, because the defaults cover most common use cases. This behavior makes the project file smaller and easier to understand and edit by hand, if needed.

The following table shows which elements and which globs are included and excluded in the .NET SDK:

Element	Include glob	Exclude glob	Remove glob
Compile	**/*.cs (or other language extensions)	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	N/A
EmbeddedResource	**/*.resx	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	N/A
None	**/*	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	**/*.cs; **/*.resx

> **Note**
>
> The `./bin` and `./obj` folders, which are represented by the `$(BaseOutputPath)` and `$(BaseIntermediateOutputPath)` MSBuild properties, are excluded from the globs by default. Excludes are represented by the **DefaultItemExcludes property**.

The .NET Desktop SDK has more includes and excludes for WPF. For more information, see WPF default includes and excludes.

Build errors

If you explicitly define any of these items in your project file, you're likely to get a "NETSDK1022" build error similar to the following:

> Duplicate 'Compile' items were included. The .NET SDK includes 'Compile' items from your project directory by default. You can either remove these items from your project file, or set the 'EnableDefaultCompileItems' property to 'false' if you want to explicitly include them in your project file.

> Duplicate 'EmbeddedResource' items were included. The .NET SDK includes 'EmbeddedResource' items from your project directory by default. You can either remove these items from your project file, or set the 'EnableDefaultEmbeddedResourceItems' property to 'false' if you want to explicitly include them in your project file.

To resolve the errors, do one of the following:

- Remove the explicit `Compile`, `EmbeddedResource`, or `None` items that match the implicit ones listed on the previous table.
- Set the EnableDefaultItems property to `false` to disable all implicit file inclusion:

  XML

  ```
  <PropertyGroup>
    <EnableDefaultItems>false</EnableDefaultItems>
  </PropertyGroup>
  ```

  If you want to specify files to be published with your app, you can still use the known MSBuild mechanisms for that, for example, the `Content` element.

- Selectively disable only `Compile`, `EmbeddedResource`, or `None` globs by setting the EnableDefaultCompileItems, EnableDefaultEmbeddedResourceItems, or EnableDefaultNoneItems property to `false`:

  XML

  ```
  <PropertyGroup>
    <EnableDefaultCompileItems>false</EnableDefaultCompileItems>

  <EnableDefaultEmbeddedResourceItems>false</EnableDefaultEmbeddedResourc
  eItems>
    <EnableDefaultNoneItems>false</EnableDefaultNoneItems>
  </PropertyGroup>
  ```

  If you only disable `Compile` globs, Solution Explorer in Visual Studio still shows *.cs items as part of the project, included as `None` items. To disable the implicit `None` glob, set `EnableDefaultNoneItems` to `false` too.

Implicit using directives

Starting in .NET 6, implicit global using directives are added to new C# projects. This means that you can use types defined in these namespaces without having to specify their fully qualified name or manually add a `using` directive. The *implicit* aspect refers to the fact that the `global using` directives are added to a generated file in the project's

obj directory.

Implicit `global using` directives are added for projects that use one of the following SDKs:

- `Microsoft.NET.Sdk`
- `Microsoft.NET.Sdk.Web`
- `Microsoft.NET.Sdk.Worker`
- `Microsoft.NET.Sdk.WindowsDesktop`

A `global using` directive is added for each namespace in a set of default namespaces that are based on the project's SDK. These default namespaces are shown in the following table.

SDK	Default namespaces
Microsoft.NET.Sdk	System System.Collections.Generic System.IO System.Linq System.Net.Http System.Threading System.Threading.Tasks
Microsoft.NET.Sdk.Web	System.Net.Http.Json Microsoft.AspNetCore.Builder Microsoft.AspNetCore.Hosting Microsoft.AspNetCore.Http Microsoft.AspNetCore.Routing Microsoft.Extensions.Configuration Microsoft.Extensions.DependencyInjection Microsoft.Extensions.Hosting Microsoft.Extensions.Logging
Microsoft.NET.Sdk.Worker	Microsoft.Extensions.Configuration Microsoft.Extensions.DependencyInjection Microsoft.Extensions.Hosting Microsoft.Extensions.Logging
Microsoft.NET.Sdk.WindowsDesktop (Windows Forms)	Microsoft.NET.Sdk namespaces System.Drawing System.Windows.Forms

SDK	Default namespaces
Microsoft.NET.Sdk.WindowsDesktop (WPF)	Microsoft.NET.Sdk namespaces Removed System.IO Removed System.Net.Http

If you want to disable this feature, or if you want to enable implicit `global using` directives in an existing C# project, you can do so via the ImplicitUsings MSBuild property.

You can specify additional implicit `global using` directives by adding `Using` items (or `Import` items for Visual Basic projects) to your project file, for example:

XML

```xml
<ItemGroup>
  <Using Include="System.IO.Pipes" />
</ItemGroup>
```

Implicit package references

When targeting .NET Core 1.0 - 2.2 or .NET Standard 1.0 - 2.0, the .NET SDK adds implicit references to certain *metapackages*. A metapackage is a framework-based package that consists only of dependencies on other packages. Metapackages are implicitly referenced based on the target framework(s) specified in the TargetFramework or TargetFrameworks property of your project file.

XML

```xml
<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
</PropertyGroup>
```

XML

```xml
<PropertyGroup>
  <TargetFrameworks>netcoreapp2.1;net462</TargetFrameworks>
</PropertyGroup>
```

If needed, you can disable implicit package references using the

DisableImplicitFrameworkReferences property, and add explicit references to just the frameworks or packages you need.

Recommendations:

- When targeting .NET Framework, .NET Core 1.0 - 2.2, or .NET Standard 1.0 - 2.0, don't add an explicit reference to the `Microsoft.NETCore.App` or `NETStandard.Library` metapackages via a `<PackageReference>` item in your project file. For .NET Core 1.0 - 2.2 and .NET Standard 1.0 - 2.0 projects, these metapackages are implicitly referenced. For .NET Framework projects, if any version of `NETStandard.Library` is needed when using a .NET Standard-based NuGet package, NuGet automatically installs that version.
- If you need a specific version of the runtime when targeting .NET Core 1.0 - 2.2, use the `<RuntimeFrameworkVersion>` property in your project (for example, `1.0.4`) instead of referencing the metapackage. For example, you might need a specific patch version of 1.0.0 LTS runtime if you're using self-contained deployments.
- If you need a specific version of the `NETStandard.Library` metapackage when targeting .NET Standard 1.0 - 2.0, you can use the `<NetStandardImplicitPackageVersion>` property and set the version you need.

Build events

In SDK-style projects, use an MSBuild target named `PreBuild` or `PostBuild` and set the `BeforeTargets` property for `PreBuild` or the `AfterTargets` property for `PostBuild`.

XML

```
<Target Name="PreBuild" BeforeTargets="PreBuildEvent">
    <Exec Command="&quot;$(ProjectDir)PreBuildEvent.bat&quot;
&quot;$(ProjectDir)..\&quot; &quot;$(ProjectDir)&quot;
&quot;$(TargetDir)&quot;" />
</Target>

<Target Name="PostBuild" AfterTargets="PostBuildEvent">
   <Exec Command="echo Output written to $(TargetDir)" />
</Target>
```

Note

- You can use any name for the MSBuild targets. However, the Visual Studio IDE recognizes `PreBuild` and `PostBuild` targets, so by using those names, you can edit the commands in the IDE.
- The properties `PreBuildEvent` and `PostBuildEvent` are not recommended in SDK-style projects, because macros such as `$(ProjectDir)` aren't resolved. For example, the following code is not supported:

XML

```
<PropertyGroup>
  <PreBuildEvent>"$(ProjectDir)PreBuildEvent.bat" "$(ProjectDir)..\"
"$(ProjectDir)" "$(TargetDir)"</PreBuildEvent>
</PropertyGroup>
```

Customize the build

There are various ways to customize a build. You may want to override a property by passing it as an argument to an msbuild or dotnet command. You can also add the property to the project file or to a *Directory.Build.props* file. For a list of useful properties for .NET projects, see MSBuild reference for .NET SDK projects.

Custom targets

.NET projects can package custom MSBuild targets and properties for use by projects that consume the package. Use this type of extensibility when you want to:

- Extend the build process.
- Access artifacts of the build process, such as generated files.
- Inspect the configuration under which the build is invoked.

You add custom build targets or properties by placing files in the form `<package_id>.targets` or `<package_id>.props` (for example, `Contoso.Utility.UsefulStuff.targets`) in the *build* folder of the project.

The following XML is a snippet from a *.csproj* file that instructs the dotnet pack command what to package. The `<ItemGroup Label="dotnet pack instructions">` element places the targets files into the *build* folder inside the package. The `<Target`

`Name="CollectRuntimeOutputs" BeforeTargets="_GetPackageFiles">` element places the assemblies and *.json* files into the *build* folder.

XML

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

  ...
  <ItemGroup Label="dotnet pack instructions">
    <Content Include="build\*.targets">
      <Pack>true</Pack>
      <PackagePath>build\</PackagePath>
    </Content>
  </ItemGroup>
  <Target Name="CollectRuntimeOutputs" BeforeTargets="_GetPackageFiles">
    <!-- Collect these items inside a target that runs after build but be-
fore packaging. -->
    <ItemGroup>
      <Content Include="$(OutputPath)\*.dll;$(OutputPath)\*.json">
        <Pack>true</Pack>
        <PackagePath>build\</PackagePath>
      </Content>
    </ItemGroup>
  </Target>
  ...

</Project>
```

To consume a custom target in your project, add a `PackageReference` element that points to the package and its version. Unlike the tools, the custom targets package is included in the consuming project's dependency closure.

You can configure how to use the custom target. Since it's an MSBuild target, it can depend on a given target, run after another target, or be manually invoked by using the `dotnet msbuild -t:<target-name>` command. However, to provide a better user experience, you can combine per-project tools and custom targets. In this scenario, the per-project tool accepts whatever parameters are needed and translates that into the required dotnet msbuild invocation that executes the target. You can see a sample of this kind of synergy on the MVP Summit 2016 Hackathon samples repo in the dotnet-packer project.

See also

- Install .NET Core
- How to use MSBuild project SDKs
- Package custom MSBuild targets and props with NuGet