

# Chapter 46:

## .NET Core with csproj

### What's in this Chapter?

.NET Core CLI

.NET Standard

Using Legacy Libraries

Creating Self-Contained Applications

Migration from project.json

Microsoft's Support Strategy

### Wrox.com Code Downloads for This Chapter

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/procsharp](http://www.wrox.com/go/procsharp) on the Download Code tab. The code for this chapter is divided into the following major examples:

- HelloWorld
- HelloWorldSelfContained
- XUnitTestSample
- MSTestSample

Samples from other chapters are used as well to initiate the migration process.

---

## Overview

The book Professional C# 6 and .NET Core 1.0 was released with Visual Studio 2015. Shortly after the book release, .NET Core 1.0 was released on June 27<sup>th</sup>, 2016. While .NET Core was released, the tools for .NET Core have been in a preview state. Now, with the release of Visual Studio 2017, the .NET Core tools are released as well. While the source code for .NET Core did not change, the project files changed.

To give you all the information what changed to make your way through .NET Core, this Chapter gives you all the information you need with the new tools. There's some overlap with information you can already read in

the book. However, with the release of the tool it helps having a fresh view and not only include information what's changed.

All the samples for the book have been updated to Visual Studio 2017. At the GitHub repository of the book (<https://www.github.com/ProfessionalCSharp/ProfessionalCSharp6>) you can get the Visual Studio 2015 version of the projects in the VS2015 branch, and the Visual Studio 2017 version in the VS2017 and Master branches.

The book described the project file `project.json`, as this was the project file used with Visual Studio 2015. Instead of using this file with JSON syntax, a switch was made back to XML with a `csproj` file to support *MSBuild*. Previously the MSBuild project file was very long and not easy to work with an editor. This has changed. Now a lot of defaults are defined, which reduce the length of the file and make it easier to work with. For example, instead of the need to add every source file of the project explicitly, now by default all C# files from the directory are added automatically.

## Note

Why did the project file change? When .NET Core was created, MSBuild was only available on Windows, not on Linux or other platforms. Because of this, a custom build engine was created for .NET Core. A new build engine making use of `project.json` was created. This project file was using JSON with similarities to other project files, e.g. npm (Node Package Manager) with `package.json` for scripting packages.

Using the `project.json` based build engine also had some disadvantages. While older legacy projects (Windows Forms, WPF...) still were based on the XML syntax of `csproj` files. Thus, some great tools of Visual Studio available for legacy applications didn't support the JSON syntax of `project.json`.

Over time, MSBuild was open sourced, and is now available on Linux as well. This opened the door to use this mature and full capable build engine with .NET Core.

However, because the `csproj` syntax was way too much babbling, and thus was not meant to write this file by hand, the syntax had to be simplified. During the previews of the new `csproj` syntax, the project file was shortened build after build. Now we have a practically short version of this file that can also be created by hand.

Now, instead of using the .NET Core CLI tools from Visual Studio and Visual Studio Code, a shared SDK component is available that defines the build commands. The .NET Core CLI tools, as well as Visual Studio and Visual Studio Code make use of this shared component. The repository of the shared component is <https://github.com/dotnet/sdk>, while the .NET Core CLI tools are available at <https://github.com/dotnet/cli>.

While the progress to csproj is great, there are still some disadvantages on the new XML syntax and the current state of the tools. If you were used to add NuGet packages directly within `project.json` in the code editor, doing the same with csproj, you might miss Intellisense. For getting this you can install the Visual Studio extension Project File Tools. With an update of Visual Studio, this feature will likely be supported directly from Visual Studio.

Another issue you might get into is that not all tools of Visual Studio support the new csproj syntax yet. For example, Live Unit Testing, a new feature of Visual Studio 2017, does not support .NET Core yet. I'm sure this will change in the future.

If you have bad feelings against XML in favor of JSON, stay open minded. I'm comparing XML to JSON likewise to Visual Basic and C# - with open/close statements compared to curly brackets. Don't be misled by this statement. The new csproj syntax is a lot better than the old one. Also, in Visual Studio, you can edit the csproj files without unloading the project which is still needed with older csproj project types.

---

## .NET Core Versions and Microsoft's Support

Before getting more into the new project system, you need to decide which .NET Core version best fits for your environment. Are you already programming agile, offering new features for your users in a fast pace and would like to use new APIs that can be advantageous for your application, or do you prefer not to update your application that often?

Selecting the version of .NET Core for your applications depends on your needs. You need to be aware of the strategies for **Long Term Support** (LTS) and the **Current** version.

The *LTS* version has a much longer support time than the *Current* release. In order to be on a release that is supported by Microsoft and receives updates, you need to update applications written with the *Current* release more often. You also have more features available.

At the time of this writing, the version on the *LTS* release cycle is .NET Core 1.0, while the *Current* version is .NET Core 1.1. The following table shows simplified information with the *Current* and *LTS* releases, their release dates, the current patch versions (a patch just contains fixes but no new features), and the support end.

### .NET Core Versions

Version	Release Date	Patch Version	Support Level	Support End (simplified)
.NET Core 1.1	Nov 16, 2016	1.1.1	Current	3 months after next Current
.NET Core 1.0	June 27, 2016	1.0.4	LTS	12 months after next LTS

The *support end* column from this table gives not the complete truth, but it gives the dates as expected. To be more precise on the *LTS* support length: *LTS* is supported until 3 years after the release (which would be June 27, 2019), or 12 months after the next *LTS* release, whichever is shorter.

Assuming we get the next *LTS* release in Oct, 2017, the support of .NET Core 1.0 ends Oct 2018. In case we need to wait for the next *LTS* release until Jan 2019, support for .NET Core 1.0 ends June 27, 2019.

To be more precise on the end of support for the *Current* release: the end of support ends 3 years after the release of the *LTS* version, or 12 months after the next *LTS* version, or 3 months after the next *Current* version, whichever is shorter. With this, the longest time the *Current* release could be supported is June 27, 2019 - assuming there are no newer versions before. In case we get a new *LTS* version in Oct, 2017, the support is reduced until Oct, 2018. However, let's assume we get a new *Current* version after the next *LTS* in Feb, 2018, the end of support moves to an earlier date which is May, 2018.

In a different scenario, we get a new *Current* version in May, 2017 - then the support for .NET Core 1.1 already ends Aug, 2017.

With this information in mind, you need to decide between the features you need (and probably you get some new features with a new current version), and the support length that is needed for your projects. Depending on the versions you select you also need to select the corresponding libraries that require specific minimum .NET Core versions. For example, to use the new features of Entity Framework Core 1.1, you also need .NET Core 1.1. You cannot use Entity Framework Core 1.1 with a .NET Core 1.0 project.

## Note

Read more about actual information of LTS and Current support cycles at <https://www.microsoft.com/net/core/support>.

---

# .NET Core CLI

Now let's get into the .NET Core Command Line interface. You can get the source code and newer upcoming versions at <https://github.com/dotnet/cli>. Version 1.0 is distributed with Visual Studio 2017 - you just need to select the .NET Core workloads from the Visual Studio Installer. Without Visual Studio, and for other platforms, you can get the tools with the SDK from <https://www.microsoft.com/net/download/core>. With .NET Core installed, on a Windows system you can find it at `%ProgramFiles%\dotnet`.

While the tools with its arguments are mainly the same as in the preview (also a few changes - for the better - have been made, such as the support of templates for `dotnet new`), the output changed. Let's get into the process flow:

- `dotnet new`
- `dotnet restore`
- `dotnet build`
- `dotnet run`

## dotnet new

You can create a new project with `dotnet new`. Contrary to the previous version of this tool, starting the command `dotnet new` does not create a console application. Instead, you'll get a list of installed templates and need to specify the template. This is much more powerful, as more options are available, and you can also create your own templates for creating new applications and files.

Let's start with a simple console application. Before starting the command, create the subdirectory `HelloWorld`, and set the current directory to this subdirectory.

This command creates the project for a console application:

```
> dotnet new console --language C# --framework netcoreapp1.1
```

You can write the command without the option `--language`. C# is the default language. You can also create console applications with F#. If you do not add the `--framework` option, the version selected is *Long Term Support* (LTS), version 1.0.

## Note

For LTS and Current support, check the section .NET Core Versions and Microsoft's Support earlier in the Chapter.

In case the project shouldn't have the same name as the current directory, you can use the option `--name` and give the project a different name. Using this option, a new subdirectory is created, and the new files are stored in this subdirectory. You can also supply the option `--output` to specify an output directory.

To see a list of the options available, add the `--help` option, e.g.

```
> dotnet new console --help
```

With ASP.NET MVC Core applications, the list of options is slightly longer:

```
> dotnet new mvc --help
```

For ASP.NET MVC Core applications, you can select the authentication to use, LocalDB or SQLite for the database, and the .NET Core version:

```
ASP.NET Core Web App (C#)
Author: Microsoft
Options:
```

```
-au|--auth           The type of authentication to use
                     None             - No authentication
                     Individual        - Individual authentication
                     Default: None

-uld|--use-local-db  Whether or not to use LocalDB instead of SQLite
                     bool - Optional
                     Default: false

-f|--framework       netcoreapp1.0    - Target netcoreapp1.0
                     netcoreapp1.1    - Target netcoreapp1.1
                     Default: netcoreapp1.1
```

The result of creating a console application is the source code file `Program.cs`, and the project file `HelloWorld.csproj`. `Program.cs` is a simple *Hello, World!* application where the C# source code didn't change compared to the previous release. The project file

HelloWorld.csproj contains the new XML syntax for the build file. The file is very short as useful defaults have been defined (code file HelloWorld\HelloWorld.csproj):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>
</Project>
```

The root element of the csproj file is the `Project` element. The `Sdk` attribute is set to `Microsoft.NET.Sdk` - this defines the available and configured MSBuild tasks and targets. With Web projects, the `Sdk` attribute is set to `Microsoft.NET.Sdk.Web`.

`Microsoft.NET.Sdk.Web` contains additional defaults such as publishing settings for the `wwwroot` folder, `cshtml` files, and more.

The element `Project` contains `PropertyGroup` elements. The only `PropertyGroup` that is created within the simple Console application defines the `OutputType` of the application to be an executable, and the version of .NET Core with the alias `netcoreapp1.1`.

## Note

Selecting `netcoreapp` (or `netstandard` with libraries) with the `TargetFramework` element implicitly references the meta-packages `Microsoft.NetCore.App` or `NetStandard.Library`, so you do not need to reference these packages explicitly.

## Specifying multiple Target Frameworks

Instead of just building an application for a single framework, a csproj file can be configured to build the application for multiple frameworks. For this, you need to specify the element `TargetFrameworks` instead of `TargetFramework` and specify the frameworks with the `;` delimiter. The following snippet specifies .NET Core 1.1 and .NET Framework 4.6.2 as target frameworks:

```
<TargetFrameworks>netcoreapp1.1;net462</TargetFrameworks>
```

Specifying multiple frameworks results in multiple binaries being compiled for the application. For every target framework, a subdirectory is created for the above example.

## Adding More Templates for *dotnet new*

You can install additional templates for the `dotnet new` command. For example, to get Single Page Application (SPA) templates with Angular, Aurelia, Knockout, and React, you can start this command:

```
> dotnet new --install "Microsoft.AspNetCore.SpaTemplates::*"
```

### Note

See <https://github.com/aspnet/javascriptservices> for more information on these templates. Check the list on templates for `dotnet new` at

<https://github.com/dotnet/templating/wiki/Available-templates-for-dotnet-new>. For creating custom templates read this blog from the .NET team:

<https://blogs.msdn.microsoft.com/dotnet/2017/04/02/how-to-create-your-own-templates-for-dotnet-new/>.

## More Project Templates

The command `dotnet new --list` shows all the templates that are available for you. Examples are `classlib` to create a .NET Standard library, `xunit` to create an XUnit test project, `mstest` to create a Microsoft Test project, `web` for an empty Web application, and `mvc` for a Web application containing ASP.NET Core MVC code.

If the current directory of the command line is an already created project, the result of `dotnet new --list` is different. Here, you can create a configuration file for building a NuGet package (`nugetconfig`), a configuration file for a Web application (`webconfig`), or a solution file (`sln`).

### Note

Depending on what the current directory is, `dotnet new --list` gives a different output. Set the current directory to an empty directory to see the options for projects you can create. If you are already positioned in a directory that already contains a project file, you can only see templates that you can create in this directory.



## Note

The .NET Standard allows creating libraries that can be used from .NET Core, the .NET Framework, and Xamarin. The .NET Standard removes the need for portable libraries and is discussed later in this Chapter.

## dotnet restore

After the project has been created, `dotnet restore` downloads all NuGet packages and creates a build file.

The NuGet packages are downloaded to the users' profile (on a Windows system, the directory `%UserProfile%\ .nuget\packages`, on Linux: `~/.nuget/packages`), thus invoking the command another time, also from another project, the same files do not need to be downloaded from the NuGet server once more. Where the files are downloaded from is defined in the file `NuGet.Config`. This file resides in the roaming profile directory `%AppData%\NuGet` or `~/.nuget/NuGet/` on Linux and contains the link to the NuGet server:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json"
        protocolVersion="3" />
  </packageSources>
</configuration>
```

Depending on your configuration, different servers (or simple folders with `.nupk` files) could be added. You can also link to your own NuGet server. If you have a project that needs a different configuration than the default one, you can add a custom `NuGet.Config` to the project directory. This file located in the project directory is used instead of the default one.

## Note

Different NuGet servers are needed if you work with daily builds. For example, the daily builds of the upcoming .NET Core 2.0 are located at <https://dotnet.myget.org/gallery/dotnet-core>.

`dotnet restore` also creates MSBuild files located in the `obj` subdirectory of the project. The file with the `props` extension contains properties of the project where you can see the folder of the NuGet packages that is used, as well as the version of the NuGet tool that is used. The second MSBuild file generated ends with the `targets` file extension and

contains the build targets. Another file is generated: `project.assets.json`. In this file, you can see all the NuGet packages and projects that are referenced - including the complete hierarchical tree to show all dependencies with the version number.

## **dotnet build**

After restoring all NuGet packages, the project can be built. This is done with the command `dotnet build`. This command requires the previously mentioned `project.assets.json` file

Starting this command creates a `bin` directory for the binaries, and compiles the sources to build the binaries. By default, a *Debug* build is done where the binaries are put into the `bin/Debug` directory. You can supply the configuration with the `--configuration` option:

```
> dotnet build --configuration Debug
```

*Debug* and *Release* configurations are available by default. The generated IL code for a simple Hello, World! program differs between debug and release builds. You can verify the intermediate language (IL) code from the generated library `HelloWorld.dll` using the tool `ildasm` (.NET Framework IL Disassembler). There you can see that the debug build contains `nop` (no operation) statements in the `Main` method before and after the `WriteLine`, whereas the `nop` statements are removed from the release build.

### **Note**

You can also create custom configurations (e.g. for *staging* and *production* servers).

## **dotnet run**

With `dotnet run` you can run the application. `dotnet run` also builds the application if it wasn't built before. With the option `--configuration` you can supply to build debug or release builds

```
> dotnet run --configuration Release
```

The `dotnet run` command is not meant for the production environment. For production, a publish package should be built. This is discussed later in this Chapter.

## Note

`dotnet new`, `dotnet restore`, `dotnet build`, and `dotnet run` are the commands you typically need with a build process. More commands are available for unit testing, creating NuGet packages, migration from old project files, and more. Many additional commands are discussed in the next sections in this Chapter.

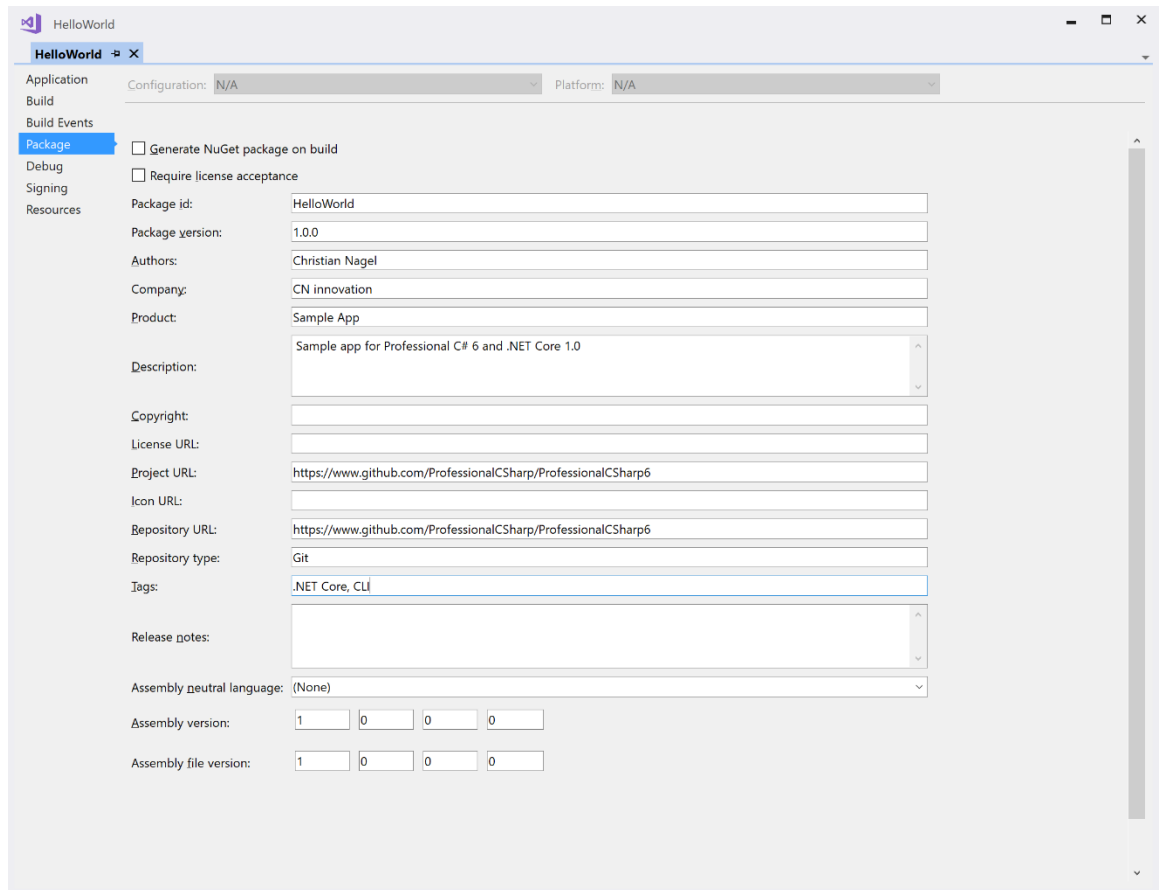
---

# Project Metadata

With the .NET Framework (and .NET Core projects created with Visual Studio 2015), project metadata is typically stored in the file `AssemblyInfo.cs` using assembly-scoped attributes. Attribute types typically used are `AssemblyTitle`, `AssemblyDescription`, `AssemblyCompany`, `AssemblyProduct`, `AssemblyCopyright` and `AssemblyTrademark`. This is still possible with the new project format. If the assembly attributes should be used, generating this information from MSBuild needs to be disabled by setting the corresponding MSBuild elements. A few examples of these elements are `GenerateAssemblyTitleAttribute`, `GenerateAssemblyDescriptionAttribute`, `GenerateAssemblyCompanyAttribute` that can be set to `false`.

```
<PropertyGroup>
  <TargetFramework>netcoreapp1.0</TargetFramework>
  <AssemblyName>SimpleArrays</AssemblyName>
  <OutputType>Exe</OutputType>
  <GenerateAssemblyTitleAttribute>>false</GenerateAssemblyTitleAttribute>
  <GenerateAssemblyDescriptionAttribute>>false
</GenerateAssemblyDescriptionAttribute>
  <GenerateAssemblyConfigurationAttribute>>false
</GenerateAssemblyConfigurationAttribute>
  <GenerateAssemblyCompanyAttribute>>false</GenerateAssemblyCompanyAttribute>
  <GenerateAssemblyProductAttribute>>false</GenerateAssemblyProductAttribute>
  <GenerateAssemblyCopyrightAttribute>>false
</GenerateAssemblyCopyrightAttribute>
</PropertyGroup>
```

Defining project metadata, you can define the settings with the Package tab in the project properties of Visual Studio (see Figure 46-1).



**Figure 46-1**

After changing the metadata with Visual Studio, HelloWorld.csproj contains the updates with Authors, Company, Product, Description, and other elements:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFrameworks>netcoreapp1.1;net462</TargetFrameworks>
    <Authors>Christian Nagel</Authors>
    <Company>CN innovation</Company>
    <Product>Sample App</Product>
    <Description>Sample app for Professional C# 6 and .NET Core 1.0
  </Description>
    <Copyright />
    <RepositoryUrl>
      https://www.github.com/ProfessionalCSharp/ProfessionalCSharp6
    </RepositoryUrl>
    <RepositoryType>Git</RepositoryType>
    <PackageTags>.NET Core, CLI</PackageTags>
    <PackageProjectUrl>
      https://www.github.com/ProfessionalCSharp/ProfessionalCSharp6
    </PackageProjectUrl>
  </PropertyGroup>
```

</Project>

---

## Working with Solutions

The release of .NET Core tools allow working with solution files. While Visual Studio supports working with solutions for a long time, creating and managing solution files was not available with the .NET Core CLI tools in the pre-release version.

Visual Studio 2017 allows working with folders instead of using solution files, so this enhancement to the .NET Core CLI tools wouldn't really be necessary. However, working with solutions instead of folders gives more flexibility. You can create solution files that span multiple projects independent of the folder structure, e.g. a solution that contains all API services with their dependencies, and a different solution for the client projects. Another solution can be created to contain client- and server projects for a specific feature of the application. One project can be contained in multiple solutions.

Passing the template name `sln` to `dotnet new` creates a solution file:

```
> dotnet new sln --name SampleSolution
```

The generated solution file `SampleSolution.sln` only contains global definitions. From the content it's clear this is geared toward Visual Studio:

```
Microsoft Visual Studio Solution File, Format Version 12.00
# Visual Studio 15
VisualStudioVersion = 15.0.26124.0
MinimumVisualStudioVersion = 15.0.26124.0
Global
    GlobalSection(SolutionConfigurationPlatforms) = preSolution
        Debug|Any CPU = Debug|Any CPU
        Debug|x64 = Debug|x64
        Debug|x86 = Debug|x86
        Release|Any CPU = Release|Any CPU
        Release|x64 = Release|x64
        Release|x86 = Release|x86
    EndGlobalSection
    GlobalSection(SolutionProperties) = preSolution
        HideSolutionNode = FALSE
    EndGlobalSection
EndGlobal
```

To add and remove projects, the `dotnet sln` command can be used. Here, the *HelloWorld* project is added to the solution *SampleSolution*. You need to supply the filenames for the project. The solution file can be

omitted if the command is invoked from the same directory as the solution file:

```
> dotnet sln SampleSolution.sln add HelloWorld\HelloWorld.csproj
```

Now the project is added as you can see in the solution file `SampleSolution.sln`. A GUID uniquely identifies the project. This GUID is used later in the file for the different configurations:

```
Microsoft Visual Studio Solution File, Format Version 12.00
# Visual Studio 15
VisualStudioVersion = 15.0.26124.0
MinimumVisualStudioVersion = 15.0.26124.0
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "HelloWorld",
    "HelloWorld\HelloWorld.csproj", "{03E79868-A4AB-4016-AE03-3CAA89C6542A}"
EndProject
Global
    GlobalSection(SolutionConfigurationPlatforms) = preSolution
        Debug|Any CPU = Debug|Any CPU
        Debug|x64 = Debug|x64
        Debug|x86 = Debug|x86
        Release|Any CPU = Release|Any CPU
        Release|x64 = Release|x64
        Release|x86 = Release|x86
    EndGlobalSection
    GlobalSection(SolutionProperties) = preSolution
        HideSolutionNode = FALSE
    EndGlobalSection
    GlobalSection(ProjectConfigurationPlatforms) = postSolution
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Debug|Any CPU.ActiveCfg = Debug|Any CPU
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Debug|Any CPU.Build.0 = Debug|Any CPU
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Debug|x64.ActiveCfg = Debug|x64
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Debug|x64.Build.0 = Debug|x64
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Debug|x86.ActiveCfg = Debug|x86
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Debug|x86.Build.0 = Debug|x86
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Release|Any CPU.ActiveCfg = Release|Any CPU
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Release|Any CPU.Build.0 = Release|Any CPU
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Release|x64.ActiveCfg = Release|x64
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Release|x64.Build.0 = Release|x64
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Release|x86.ActiveCfg = Release|x86
        {03E79868-A4AB-4016-AE03-3CAA89C6542A}.Release|x86.Build.0 = Release|x86
    EndGlobalSection
EndGlobal
```

Other `dotnet sln` commands are `remove` to remove a project, and `list` to list all projects for a solution.

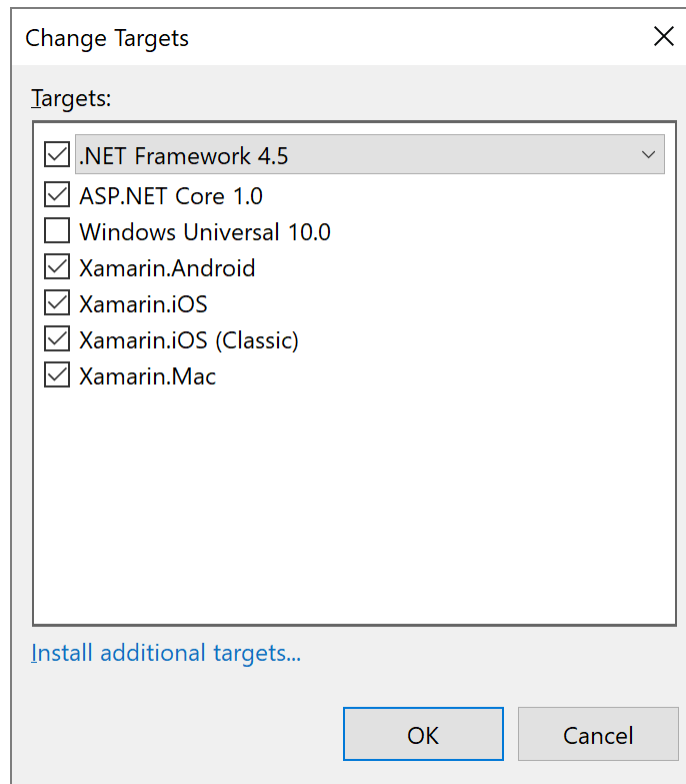
---

## Creating and using Libraries and .NET Standard

To create libraries, you have some more options you can choose from. When creating **.NET Framework libraries**, they can only be used from .NET Framework applications such as WPF, Windows Forms applications, or Web applications with ASP.NET.

This is similar with **.NET Core libraries**: when creating .NET Core libraries, they can only be used from .NET Core applications. To create libraries that can be used by multiple platforms, e.g. Silverlight, WPF, Xamarin, and UWP, portable libraries were useful.

If you create a **portable library**, you need to specify the platform target including the version number you need to support (see Figure 46-2). The more options you select, and the older versions of these options, the less APIs are available.



**Figure 46-2**

The APIs available need to be defined with a matrix. Every change of a platform target changes APIs. When one platform is added to the list, it's not enough to add a simple list of APIs the platform supports. Thus, only the smallest set of APIs available on each platform can be used. This is a burden for Microsoft, and adds complexity creating portable libraries that itself make use of portable libraries. You cannot reference another portable library that targets a larger set of APIs than the library you are creating.

Now there's a replacement for portable libraries - the **.NET Standard**. Instead of having a matrix that defines the APIs available, with every

version of the .NET Standard only APIs are added, but none removed. So, for Microsoft it's a lot easier to define the standard, and it's a lot easier to work with it. Let's get into more details on the .NET Standard, on creating .NET Standard libraries, and on using legacy libraries with .NET Core.

## **.NET Standard**

.NET Standard allows creating libraries that can be shared between different platforms. Creating a library targeting *.NET Standard 1.4* allows using this library from .NET Core applications, with .NET Framework 4.6.1 and later, from the Universal Windows Platform (UWP), Xamarin.Android 7.0, Xamarin.iOS 10.0, and Mono 4.6.

If Windows 8.1 needs to be supported, you can only use the APIs available in the *.NET Standard 1.2*. The .NET Standard 1.2 is also supported by .NET Framework 4.5.1.

The higher the version number of the .NET Standard, the more APIs are available. The lower the version, the more platforms can use the library.

See <https://docs.microsoft.com/en-us/dotnet/articles/standard/library> for a detailed list on the .NET Standard support.

### **Note**

For a complete list about what API is available on which platform, and listed in which .NET Standard version, check <https://apisof.net>. With this list, you can see for example that `List<T>` from the `System.Collections.Generic` namespace is available in the .NET Framework since 2.0, and with every other platform. It's part of the .NET Standard since 1.0. With .NET Core 1.0 and 1.1 it's in version 4.0.10.0 of the assembly `System.Collections`, and with .NET Core 2.0 the `System.Collections` assembly changes to version 4.1.0.0. You can also see the usage in apps. `List<T>` is used in 63.9% of apps discovered from API Port Telemetry.

### **Note**

The next version of the .NET Standard - .NET Standard 2.0 - defines many APIs not available with .NET Core 1.1, but already available with .NET Framework 4.6.1. .NET Core 2.0 will implement this standard as well. This makes it easier to move existing .NET Framework applications to the new world of .NET Core.



## **Creating .NET Standard Libraries**

Using the command `dotnet new`, you can create .NET Standard libraries, and .NET Core libraries. Invoking

```
> dotnet new lib
```

creates a library for the .NET Standard 1.4, as you can see from the project file:

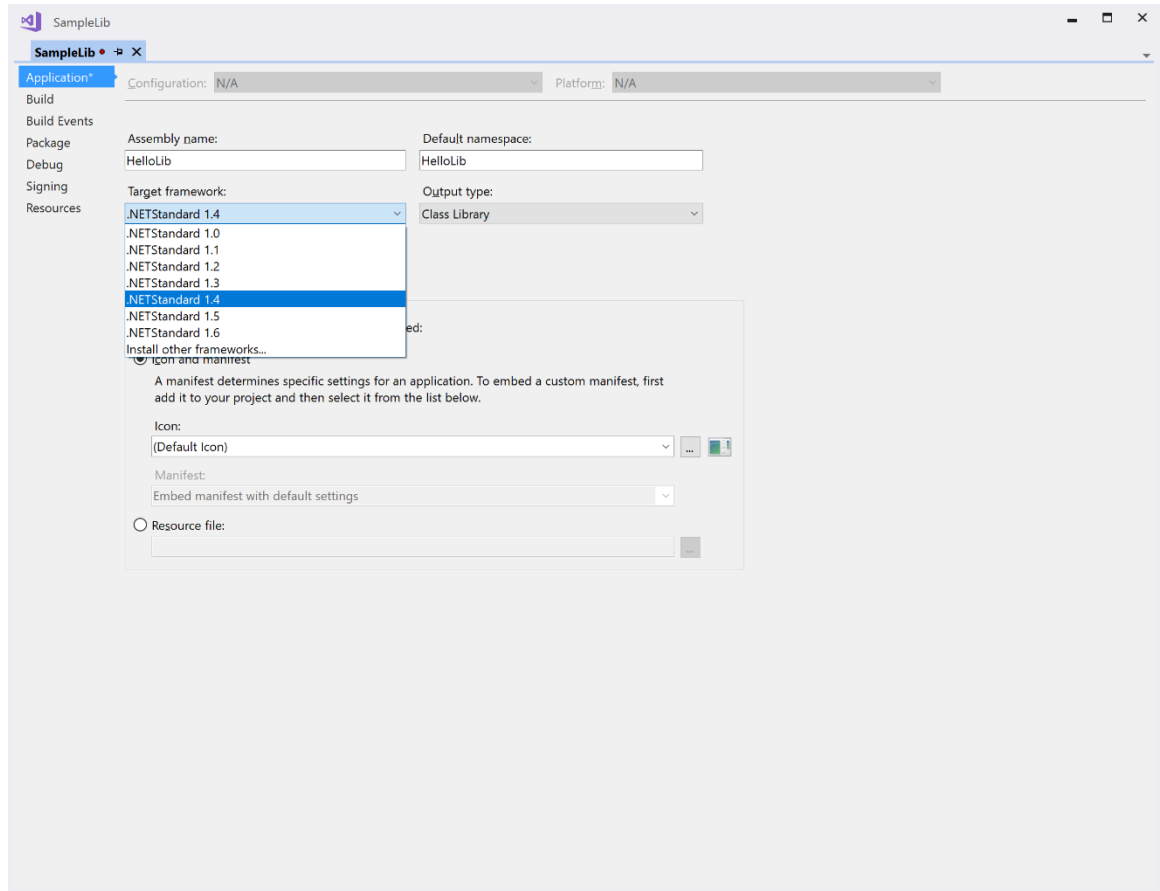
```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard1.4</TargetFramework>
  </PropertyGroup>

</Project>
```

Setting the `--framework` option with the `dotnet new lib` command, you can define any .NET Standard to use (`netstandard1.0` to `netstandard1.6`), or build a library for .NET Core (with the options `netcoreapp1.0` and `netcoreapp1.1`).

Using Visual Studio 2017, in the project templates you have a Class Library (.NET Standard) and a Class Library (.NET Core) available. With the tab Application of the project settings, you can select your desired target framework (see Figure 46-3).



**Figure 46-3**

## Note

What's the reason to build a .NET Core Library instead of a .NET Standard Library? The .NET Standard includes APIs that are available on every platform, are mature enough to not be updated frequently, and are widely used. See the documentation about the .NET Standard Review Body about the criteria for the APIs coming to the standard: <https://github.com/dotnet/standard/blob/master/docs/review-board/README.md>. Contrary to that, new APIs are coming to .NET Core first, before they will be available in a future version of the standard. If you need to use APIs from your libraries that are only available for .NET Core, create a .NET Core library. Be aware that this library can only be used from .NET Core applications. Creating libraries that are supposed to be used by as many platforms as possible, use the .NET Standard.

## Using Legacy Libraries

Not all important NuGet packages are yet available for .NET Standard. However, many of the packages still work with .NET Core and .NET Standard projects. For example, you can put view-model types for a WPF, UWP, and Xamarin app in a .NET Standard library (instead of using a portable library). However, adding the NuGet package Prism.Core results in a failing package restore with the error: *Package restore failed. Rolling back package changes*. Getting into the details of the error in the Output window of Visual Studio, you can see this information: *Package Prism.Core 6.3.0 is not compatible with netstandard1.4*. Alongside this error, you can also see a list of platforms that are supported by this package. Among this list is a portable library with this identification: *portable-monoandroid10+monotouch10+net45+win+win81+wp8+wpa81+xamarinios10*. You can use this identification, and assign it to the `PackageTargetFallback` element.

Using this element, you can now add the NuGet package Prism.Core to the library (file `HelloLib/HelloLib.csproj`):

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

  <PropertyGroup>
    <TargetFramework>netstandard1.4</TargetFramework>
    <PackageTargetFallback>
      $(PackageTargetFallback) ;
      portable-monoandroid10+monotouch10+net45+win+win81+wp8+wpa81+xamarinios10
    </PackageTargetFallback>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Prism.Core" Version="6.3.0" />
  </ItemGroup>

</Project>
```

### Note

Using the `PackageTargetFallback` element you let the NuGet package manager ignore the dependency and let it do its work. If the application can really work with this dependency - you need to test. When all the NuGet packages moved to .NET Standard, this element is no longer needed.

`PackageTargetFallback` is a replacement for the `import` element from `project.json`.

## Note

You can read more about view-models and the MVVM pattern in Chapter 31, “Patterns with XAML Apps.”

---

# Unit Testing

Unit testing is a built-in feature of the .NET Core CLI tools. The two testing frameworks coming out of the box are *XUnit* and *MSTest*.

A unit testing project using XUnit is created with the `dotnet new xunit` command:

```
> dotnet new xunit
```

The generated project file contains references to `Microsoft.NET.Test.Sdk`, `xunit`, and `xunit.runner.visualstudio` (file `XUnitTestSample/XUnitTestSample.csproj`):

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

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.0.0" />
    <PackageReference Include="xunit" Version="2.2.0" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
  </ItemGroup>

</Project>
```

The template also creates a unit test with the `Fact` attribute (code file `XUnitTestSample/UnitTest1.cs`):

```
public class UnitTest1
{
    [Fact]
    public void Test1()
    {
    }
}
```

After the test is written, the packages restored (`dotnet restore`), and the test program built (`dotnet build`), you can run the test:

```
> dotnet test
```

## Note

Microsoft is using XUnit as testing framework for .NET Core, because MSTest was not ready for .NET Core when .NET Core was initially built - but XUnit was.

To create unit tests with MSTest, you can use `dotnet new mstest`:

```
> dotnet new mstest
```

The result is similar to the XUnit sample. The generated project file just contains references to `MSTest - Microsoft.NET.Test.Sdk`, `MSTest.TestAdapter`, and `MSTest.TestFramework` (code file `MSTestSample/MSTestSample.csproj`):

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

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.0.0" />
    <PackageReference Include="MSTest.TestAdapter" Version="1.1.11" />
    <PackageReference Include="MSTest.TestFramework" Version="1.1.11" />
  </ItemGroup>

</Project>
```

Attributes used by MSTest are `TestClass` and `TestMethod` (code file `MSTestSample/UnitTest1.cs`):

```
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
    }
}
```

The remaining process is the same, and the test runs with `dotnet test`.

## Note

You can read more about unit tests in Chapter 19, “Testing.”

---

# Using Tools

The .NET Core CLI tools can be enhanced. You've already seen enhancements with custom templates for the `dotnet new` command, but you can also add other commands. For example, the Entity Framework Core team created custom commands for the migration, and the ASP.NET team created commands for managing user secrets, and bundling of files.

## Note

The code sample for adding Entity Framework Core tools and adding user secrets is not in the source code download of this chapter. Instead, you can find these samples with the Entity Framework Core samples in Chapter 38 and the ASP.NET Core samples in Chapter 40.

## Entity Framework Core Tools

To add tools to projects, you need to add `DotnetCliToolReference` elements to the project file. For adding the Entity Framework Core tools, reference the NuGet package `Microsoft.EntityFrameworkCore.Tools.Dotnet`: (Chapter 38 project file `EntityFrameworkSamples/MenuSample/MenuSample.csproj`):

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore"
      Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
      Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
      Version="1.1.1" />
  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference
      Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="1.0.0" />
  </ItemGroup>

</Project>
```

## Note

Don't be confused by the version number of the Entity Framework Core tools. The sample code makes use of `Microsoft.EntityFrameworkCore` version 1.1.1, but the tools `Microsoft.EntityFrameworkCore.Tools.Dotnet` is at version 1.0.0. The reason for this version mismatch is that the dotnet tools have just been released, and the EF Core tools align to the versioning of the dotnet tools. Most likely, the tools and .NET Core versioning will align within the 2.0 timeframe.

With the Entity Framework Core tools referenced from the project, you can use the command `dotnet ef`. From here, the subcommands `database`, `dbcontext`, and `migrations` are available. Adding EF migrations to the `MenuCards` entity from the EF Core sample is done with

```
> dotnet ef migrations add InitMenuCards
```

The commands for the EF Core tools didn't change with the release of the tools.

## Note

You can read more about the Entity Framework Core migration tools in Chapter 38, "Entity Framework Core."

## User Secrets Tools

The tools to manage user secrets are installed to the project by referencing `Microsoft.Extensions.SecretManager.Tools` (Chapter 40 project file `WebSampleApp/WebSampleApp.csproj`):

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools"
    Version="1.0.0" />
</ItemGroup>
```

With this extension to the dotnet tools, you can invoke `dotnet user-secrets` with the commands `set`, `remove`, `list`, and `clear` to manage the user secrets.

## Note

Read information about configuration with user secrets in Chapter 40, "ASP.NET Core."

---

# Creating Packages

Using the `dotnet pack` command, and also using Visual Studio 2017, you can create NuGet packages.

Using the command line, similar to the other commands, the current directory needs to be set to the directory where the `csproj` is placed. Invoke the `dotnet pack` command for the `HelloLib` library to create a NuGet package. With this command, the file `HelloLib.1.0.0.nupkg` is created in the `bin/debug` directory. You can change the configuration with the `--configuration` option, include PDB and source files with `--include-source`, and include symbols with `--include-symbols`.

## Note

In case you need more options not directly available from the `dotnet pack` command, you can pass parameters from the `dotnet` tools directly to the `MSBuild` command. Check `dotnet msbuild -h` for more information.

In Visual Studio 2017, with the project properties in the Package tab, you can select the option Generate NuGet package on build to create the package from Visual Studio, or select Pack from the context menu of the project.

---

# Publishing

.NET Core applications can be published as small packages requiring the runtime already be installed on the target system (**framework dependent** or **portable deployment**), or they can be installed **self-contained** in which case the runtime does not need to be deployed on the target system.

A framework dependent deployment has the advantage that multiple apps can use the same .NET Core installation. This reduces the size of the app and the overall disk size needed. However, you need to have the same



version of the runtime that's used by the application installed on the system. This requirement is similar to what we have with the .NET Framework.

## Note

.NET Core assemblies use a common PE file format for executables and libraries which are independent of the operating system. This allows running your .NET Core application on different platforms without the need to compile multiple binaries. That's why *framework-dependent deployment* is also known as using a **portable** application.

On the other hand, self-contained applications have the advantage that the .NET Core runtime does not need to be installed on the target system. The runtime is distributed alongside the application. This increases the size of the app and also increases the overall disk size needed on the system.

## Framework Dependent Deployment

Let's start with the default configuration - framework dependent or portable deployment. Using this method, the runtime needs to be installed on the target system. You can get the runtimes for the different platforms at <https://www.microsoft.com/net/download/core#/runtime>.

To publish your app, you just need to run the command `dotnet publish`. For the release build, start this command with the `--configuration` option:

```
> dotnet publish --configuration Release
```

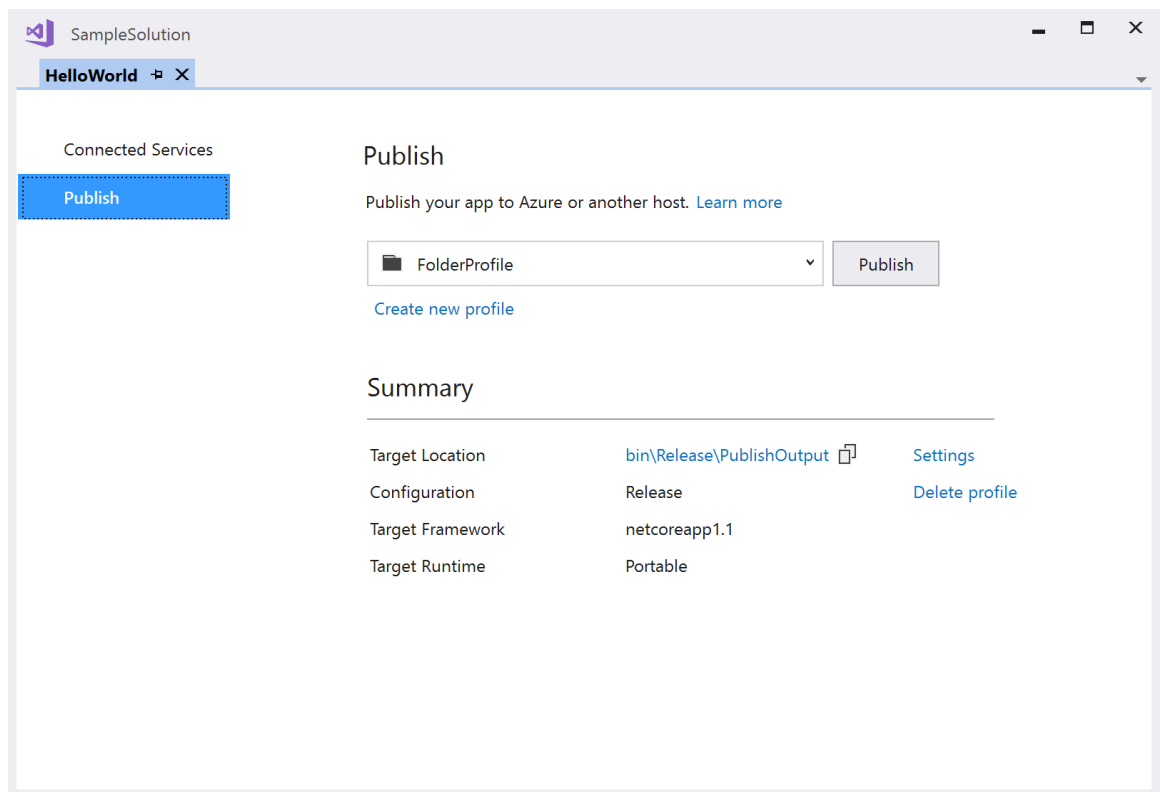
This command creates a `publish` subdirectory in the subdirectory `bin/Release/netcoreapp1.1/` containing a binary (with the DLL file extension), a PDB file containing symbols for application analysis, and a runtime configuration with `runtimeOptions`:

```
{
  "runtimeOptions": {
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "1.1.1"
    }
  }
}
```

The command `dotnet run` is meant for development purposes only. To run the application on the production system, you can invoke the `dotnet` command passing the name of the executable:

```
> dotnet HelloWorld.dll
```

Instead of using the command line tools, you can use **Build | Publish** from within Visual Studio 2017. With the dialog opened (see Figure 46-4), you just need to click the **Publish** button. By clicking on **Settings**, you can change the configuration, the target framework, and the directory where the files should be published to (see Figure 46-5). With Visual Studio, the runtime is listed as *Portable* for framework-dependent deployments.



**Figure 46-4**

**Figure 46-5**

## **Self-Contained Applications**

To create self-contained applications, you need to add `RuntimeIdentifiers` to the project file. The executable to launch the application, as well as the runtime that is included with the application is different based on the platform. The following runtime identifiers `win10-x64`, `osx.10.11-x64`, and `Ubuntu.16.10-x64` are for 64-bit Windows 10, OS X 10.11, and Ubuntu 16.10 (project file `HelloWorldSelfContained/HelloWorld.csproj`):

```
<PropertyGroup>
  <!-- ... -->
  <RuntimeIdentifiers>
    win10-x64;osx.10.11-x64;ubuntu.16.10-x64
  </RuntimeIdentifiers>
</PropertyGroup>
```

### **Note**

To get a list of all available runtime identifiers, check <https://docs.microsoft.com/en-us/dotnet/articles/core/rid-catalog> for Windows, Red Hat, Ubuntu, CentOS, Debian, Fedora, OpenSUSE, Oracle Linux and OS X.

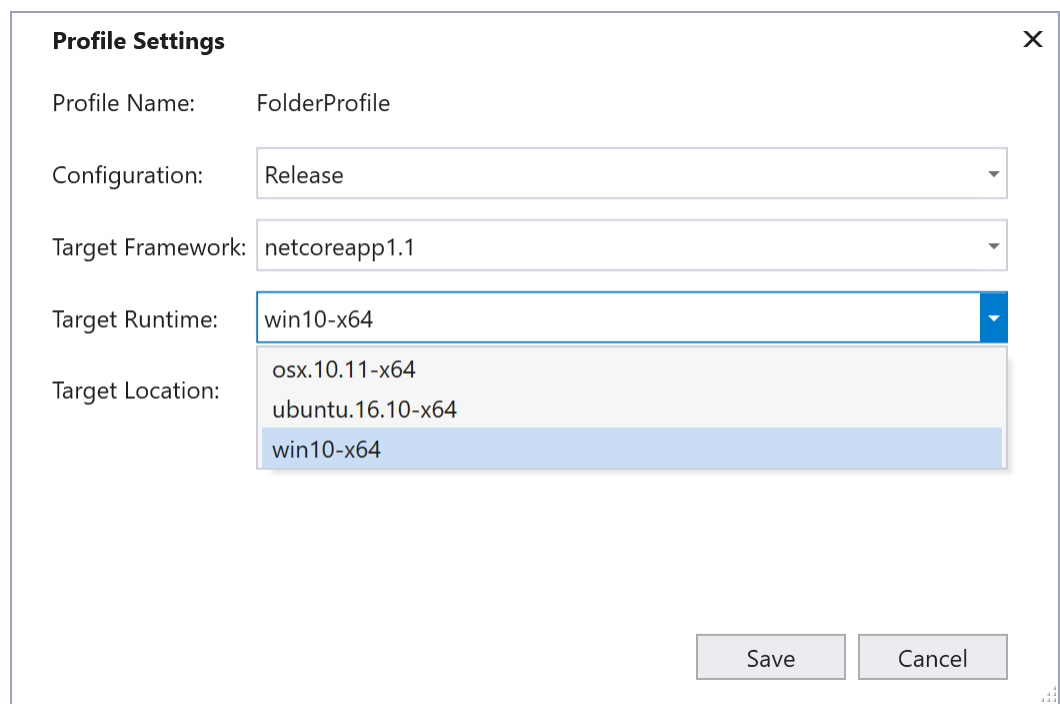
Now you can create files for publishing using the `--runtime` option, building binaries for every platform one after another:

```
> dotnet publish --runtime win10-x64 --configuration Release
> dotnet publish --runtime osx.10.11-x64 --configuration Release
> dotnet publish --runtime Ubuntu.16.10-x64 --configuration Release
```

After starting the `dotnet publish` command, you can find the files you need to publish in

`Release/{TargetFramework}/{RuntimeIdentifier}/publish` directories. Every one of these directories contains more than 100 files needed for the application including its runtime.

To create publish files from Visual Studio 2017, you can use **Build | Publish** after adding the runtime identifiers to the project file. The Settings dialog of the Publish dialog now lists the available runtimes specified within the project file as shown in Figure 46-6.



**Figure 46-6**

---

# Migration from project.json

You might already have .NET Core projects with project.json project files. Instead of creating the project files in the new format manually, you can migrate these files to the new syntax either with the .NET Core CLI tools or by using Visual Studio 2017. Both variants are covered in this Chapter. Let's start with the .NET Core CLI tools.

## Migration with .NET Core CLI Tools

For the first migration, let's copy the Entity Framework Core samples from the vs2015 branch of Chapter 37 of the book. This solution contains both .NET Framework as well as .NET Core projects. For a migration, you just need to open the command prompt, set the current directory to the directory of the solution, and start the command

```
> dotnet migrate EntityFrameworkSamples.sln
```

When using `dotnet migrate` without arguments, the current directory is searched recursively for `project.json` files to migrate. Here, a solution file was passed which results in migration of every project of the solution, and the solution file is migrated as well. You can also specify a `global.json` file which results in migration of every directory that is defined in this file. To migrate just a single project, reference the `project.json` file.

For every project that is migrated, you can see if the migration succeeded, and finally such a summary like this is shown:

```
Summary
Total Projects: 7
Succeeded Projects: 7
Failed Projects: 0
```

```
The project migration has finished. Please visit https://aka.ms/coremigration
to report any issues you've encountered or ask for help.
Files backed up to
C:\githubvs2015\ProfessionalCSharp6\EntityFramework\EntityFrameworkSamples\back
up\
```

The generated backup directory contains the original files that have been changed, so you can go back. Let's have a look how the migration was done. With the `BooksSample`, this is the original `project.json`: containing references to the NuGet packages `Microsoft.EntityFrameworkCore` and `Microsoft.EntityFrameworkCore.SqlServer`, as well as

imports for dnxcore50 which was needed to reference older .NET Core pre-released .NET Core libraries:

```
{
  "version": "1.0.0-*",
  "buildOptions": {
    "emitEntryPoint": true
  },

  "dependencies": {
    "Microsoft.EntityFrameworkCore": "1.0.1",
    "Microsoft.EntityFrameworkCore.SqlServer": "1.0.1"
  },

  "frameworks": {
    "netcoreapp1.0": {
      "imports": [ "dnxcore50" ],
      "dependencies": {
        "Microsoft.NETCore.App": {
          "type": "platform",
          "version": "1.0.1"
        }
      }
    }
  }
}
```

The generated project file BooksSample.csproj looks like this:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp1.0</TargetFramework>
    <AssemblyName>BooksSample</AssemblyName>
    <OutputType>Exe</OutputType>
    <PackageId>BooksSample</PackageId>
    <PackageTargetFallback>$(PackageTargetFallback);dnxcore50</PackageTargetFallback>
    <RuntimeFrameworkVersion>1.0.4</RuntimeFrameworkVersion>
    <GenerateAssemblyTitleAttribute>>false</GenerateAssemblyTitleAttribute>
    <GenerateAssemblyDescriptionAttribute>>false</GenerateAssemblyDescriptionAttribute>
    <GenerateAssemblyConfigurationAttribute>>false</GenerateAssemblyConfigurationAttribute>
    <GenerateAssemblyCompanyAttribute>>false</GenerateAssemblyCompanyAttribute>
    <GenerateAssemblyProductAttribute>>false</GenerateAssemblyProductAttribute>
    <GenerateAssemblyCopyrightAttribute>>false</GenerateAssemblyCopyrightAttribute>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore"
      Version="1.0.3" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
      Version="1.0.3" />
  </ItemGroup>

</Project>
```

The version of the .NET Core as well as the NuGet packages can still be updated - of course this was not done by the migration. You might decide to stick with the LTS version, or to switch to the Current version of .NET Core. LTS and Current versions have been covered in the section .NET Core Versions and Microsoft's Support earlier in this Chapter.

From the `import` element, the `PackageTargetFallback` element gets generated. For this project, this fallback can be removed as all referenced packages are now directly available for .NET Core. See the section *Use Legacy Libraries* earlier in this Chapter for more information on the `PackageTargetFallback` element.

The `AssemblyName` and the `PackageId` can be removed as well. There's a nice default if your project file has the same name as the assembly.

The `RuntimeFrameworkVersion` element can be removed as well - if you are fine with using the latest runtime version of the target framework (`TargetFramework` element) that's installed on your system which is referenced from the `Sdk` attribute of the `Project` element. You just need to specify a different `RuntimeFrameworkVersion` if your project should use a specific runtime version.

## Note

### Check the folder

`%ProgramFiles%\dotnet\shared\Microsoft.NetCore.App` to see which .NET Core runtime versions you've installed on your system.

All the elements starting with `GeneratedAssembly...` can be removed as well - if you remove the file `AssemblyInfo.cs` with the assembly attributes as was discussed in the section *Project Metadata* earlier in this Chapter.

With the manual simplification, the update to .NET Core 1.1, and the update of the referenced NuGet packages, the new version of the project file looks like this:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
    <AssemblyName>BooksSample</AssemblyName>
    <OutputType>Exe</OutputType>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore"
      Version="1.1.1" />
  </ItemGroup>
</Project>
```

```

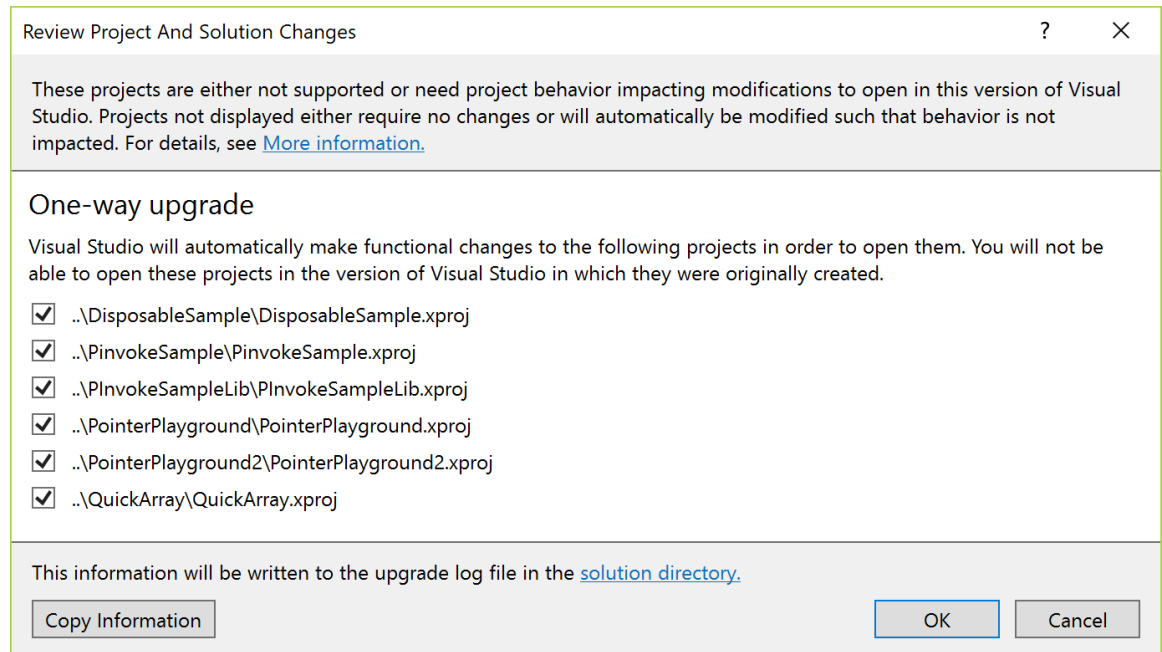
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
      Version="1.1.1" />
  </ItemGroup>

</Project>

```

## **Migration with Visual Studio 2017**

With Visual Studio 2017 you just need to open a solution to start the migration. As soon as you open a solution containing project.json files you'll see a dialog as shown in Figure 46-7. You see the One-way upgrade with special mentioning that the migrated solution cannot be opened with earlier versions of Visual Studio. Using this method, backup files are created as well, so you've got a way back in case of any issues.

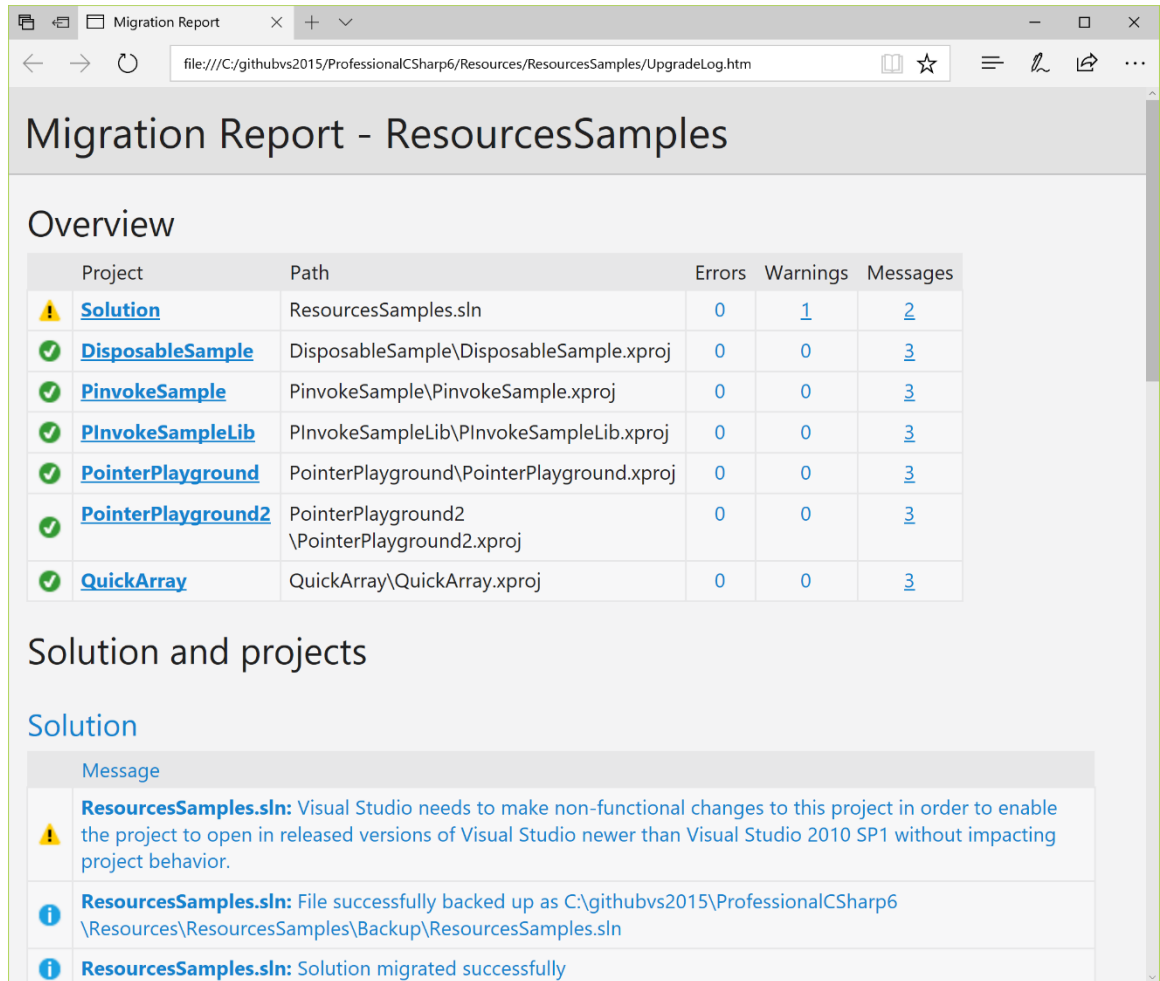


**Figure 46-7**








After the migration is completed, you can see the result as shown in Figure 46-8. The warning message for the solution contains information concerning the compatibility of the solution with older versions of Visual Studio. In order to work with csproj-based .NET Core projects, Visual Studio 2017 or newer is required anyway. The information messages just give an information about the backup of the original files, and successful migrations. As you check the generated csproj files, you can convert the files to use newer versions of .NET Core and update NuGet packages in the same way as you've seen it with the command line tools. You just can use the Application tab of the Project Properties to select the version of the



.NET Core framework, and use the NuGet Package Manager to update the NuGet packages.



The screenshot shows a web browser window titled "Migration Report" with the address bar displaying a file path. The main heading is "Migration Report - ResourcesSamples". Below this is an "Overview" section containing a table with columns for Project, Path, Errors, Warnings, and Messages. The table lists the solution and several projects, all with 0 errors and 0 warnings. Below the table is a "Solution and projects" section, followed by a "Solution" section containing a list of messages. The messages indicate that Visual Studio needs to make non-functional changes, the solution was backed up, and the migration was successful.

Project	Path	Errors	Warnings	Messages
 <a href="#">Solution</a>	ResourcesSamples.sln	0	1	<a href="#">2</a>
 <a href="#">DisposableSample</a>	DisposableSample\DisposableSample.xproj	0	0	<a href="#">3</a>
 <a href="#">PinvokeSample</a>	PinvokeSample\PinvokeSample.xproj	0	0	<a href="#">3</a>
 <a href="#">PInvokeSampleLib</a>	PInvokeSampleLib\PInvokeSampleLib.xproj	0	0	<a href="#">3</a>
 <a href="#">PointerPlayground</a>	PointerPlayground\PointerPlayground.xproj	0	0	<a href="#">3</a>
 <a href="#">PointerPlayground2</a>	PointerPlayground2\PointerPlayground2.xproj	0	0	<a href="#">3</a>
 <a href="#">QuickArray</a>	QuickArray\QuickArray.xproj	0	0	<a href="#">3</a>

**Solution and projects**

**Solution**




-  **ResourcesSamples.sln:** Visual Studio needs to make non-functional changes to this project in order to enable the project to open in released versions of Visual Studio newer than Visual Studio 2010 SP1 without impacting project behavior.
-  **ResourcesSamples.sln:** File successfully backed up as C:\githubvs2015\ProfessionalCSharp6\Resources\ResourcesSamples\Backup\ResourcesSamples.sln
-  **ResourcesSamples.sln:** Solution migrated successfully

Figure 46-8

---

## Summary

With the release of the .NET Core command line tools, .NET Core moves into a more mature state - not only the framework is released, but the tools as well. The source code of the applications doesn't need to change with the release of the tools, but the project files do need to be updated.

You've seen that the migration is a smooth process either with `dotnet migrate` or with using Visual Studio 2017 to change `project.json` files to the XML format for `csproj`.

The .NET Core CLI tools changed their output, but the commands are very similar to the preview versions. You're using `dotnet new`, `dotnet restore`, `dotnet build`, and `dotnet run`. Small but useful changes happened such as the use of templates for `dotnet new` which allow creating custom project templates. Behind the scenes of the .NET Core CLI tools a lot has changed as the tools are now based on a shared SDK component and MSBuild is used.

You've seen the features of the .NET Standard which makes Portable Libraries obsolete with easier sharing of libraries between different .NET platforms.

For publishing you've seen how can create framework dependent deployments, and self-contained applications as well as their advantages and disadvantages.

The next Chapter is about C# 7.0 - the new features of C# that changed since C# 6 and can be used with Visual Studio 2017.

## Author and Reviewers

Christian Nagel, <https://csharp.christiannagel.com>

Thanks to the reviewers Martin Ulrich and Istvan Novak