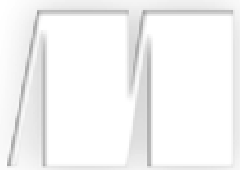
.NET Core IN ACTION

Dustin Metzgar
Foreword by Scott Hanselman

SAMPLE CHAPTER

MANNING



.NET Core in Action

by Dustin Metzgar

Sample Chapter 2

brief contents

2 Building your first .NET Core applications

This chapter covers

- Installing the .NET Core SDK
- Using the .NET CLI
- Creating and executing a .NET Core application

In this chapter, you'll learn how to set up your development environment, create an application, and deploy that application to another machine. You'll start by installing the .NET Core SDK, which includes the .NET Command-Line Interface (CLI) that's used throughout this book. From there, you'll create a console application and an ASP.NET web application. Then you'll deploy those applications.

> **NOTE FOR EARLY ADOPTERS** If you've experimented with .NET Core in the past, you may have used DNVM, DNU, or DNX. Although these tools were useful in the beginning, they had a few problems and inconsistencies. They have been deprecated in favor of the .NET CLI.

2.1 The trouble with development environments

There's something special about development environments. They accumulate a combination of tools, files, and settings that allow your application to work perfectly

during development but fail mysteriously everywhere else. Testers get frustrated when I tell them, "It works on my machine." I've been in several situations where a test was "flaky," sometimes working and sometimes not, only to discover that one of the build machines in the pool didn't have a component installed.

Making software behave consistently from development to test to production starts with the development framework. .NET Core is designed to be self-contained. It doesn't depend on Windows-specific resources like the .NET Framework—the .NET CLI is consistent on each OS. Plus, .NET Core is tailored to containers. The same container can be used for development, test, and production, reducing the friction traditionally experienced when crossing these boundaries. In the following sections, we'll explore the key features of .NET Core that produce a consistent developer experience.

2.2 Installing the .NET Core SDK

.NET Core can be installed on Windows, several Linux distros, macOS, and Docker.

An easy-to-remember URL for .NET is https://dot.net. Interestingly, you won't find much mention of the word "Core" on the .NET site. This is to clarify for newcomers to .NET that .NET Framework, .NET Standard, and .NET Core are all part of one large family. Go to the .NET site, click the Get Started button, and pick the operating system you're working on.

2.2.1 Installing on Windows operating systems

There are two methods for installing on Windows: Visual Studio and command line. Visual Studio 2017 comes with everything you'll need for .NET Core development, and the Community edition is free. It installs the .NET SDK, which has the command-line tools as well. Because the command-line tools are universal to all operating systems, this book will focus on that version. The Get Started portion on the .NET site covers both the .NET SDK and Visual Studio installations.

2.2.2 Installing on Linux-based operating systems

The process for installing .NET Core on Linux varies depending on the distro. The instructions change constantly, so by the time this book goes to print, any instructions I included here would likely be out of date. See Microsoft's .NET site (https://dot.net), click the Get Started button, choose "Linux," and pick your Linux distribution for the latest instructions.

2.2.3 Installing on macOS

.NET Core supports OS X version 10.11 and later. The best way to install is with the .pkg file available for download from the .NET site (https://dot.net). Click the Get Started button and choose "macOS."

You can also install Visual Studio for Mac, which will have the option to install the .NET Core SDK.

2.2.4 Building .NET Core Docker containers

When working with Docker, you only need to get the "dotnet" base image, which includes the .NET SDK. Simply run the following command using the Docker CLI:

```
docker run -it microsoft/dotnet:latest
```

> **GETTING STARTED WITH DOCKER** If you're not familiar with Docker, I encourage you to check out *Docker in Action* by Jeff Nickoloff (Manning, 2016). You can get started with Docker by going to https://www.docker.com/get-docker and downloading the Docker Community Edition. The Docker site includes lots of easy-to-follow documentation for installing Docker and running containers.

2.3 Creating and running the Hello World console application

The .NET Core SDK includes a sample Hello World application, and the instructions for creating it are the same on every platform. Execute the following commands on the command line or terminal:

```
mkdir hwapp
cd hwapp
dotnet new console
```

The command `dotnet new console` creates a new Hello World console application in the current folder. When new versions of .NET Core are released, it can be helpful to run this command to see if there are any updates to the fundamental pieces of .NET Core.

The `dotnet new console` command creates two files: Program.cs and hwapp.csproj. Program.cs should look similar to the following listing.

Listing 2.1 Program.cs from the Hello World application

```
using System;

namespace hwapp
{
  public class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Hello World");
    }
  }
}
```

This is a straightforward C# program. If you're familiar with C#, you'll know that the same code will work in other versions of .NET.

The hwapp.csproj file gets its name from the folder it was created in. The following listing shows the contents of this file.

Listing 2.2 hwapp.csproj from the Hello World console application

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

The csproj file describes the project. By default, all source code files in the project folder are included, including any subfolders. .NET Framework developers may be used to seeing each source code file listed explicitly in the csproj, but since the convention for .NET projects is to keep all the files under the project folder, .NET Core by default includes everything in the project folder. The `OutputType` property indicates that this project is an executable application.

2.3.1 Before you build

You now have the Hello World code and the project description, but there's a critical step that needs to take place before you build or run your application. You need to *restore* your packages. You have a set of dependencies for your project, and each dependency is a package that may also have its own dependencies. The package-restore step expands the full tree of dependencies and determines which versions of each package to install.

The command to restore packages is `dotnet restore`. Try running it to see how it works. If you're adding a new package reference to your csproj, it's a helpful command for testing whether the reference is correct.

The .NET Core SDK keeps a local cache of the packages you use. If a particular version of a package isn't in the cache, it's downloaded when you do the package restore. Since .NET Core 2.0, the .NET Core SDK will perform a restore implicitly where necessary.

2.3.2 Running a .NET Core application

When you're using the .NET Core SDK, your application will be built automatically when needed. There's no need to worry about whether or not you're executing the latest code.

Try running the Hello World application by executing `dotnet run` at the command line or terminal.

2.4 Creating an ASP.NET Core web application

Now that you've tried the Hello World console application, let's look at a Hello World ASP.NET Core application. This application will create an HTTP service that returns "Hello World" to a `GET` request.

ASP.NET is the web framework for .NET, and ASP.NET Core is a new version built from the ground up for .NET Core. In this book, I'll briefly introduce ASP.NET Core. To learn more about ASP.NET Core, check out *ASP.NET Core in Action* by Andrew Lock (Manning, 2018).

First, create a new .NET Core application. Create a new folder called hwwebapp. Execute `dotnet new web` in the hwwebapp folder. Inside the folder, you'll find the following files:

- hwwebapp.csproj
- Program.cs
- Startup.cs
- wwwroot

The hwwebapp.csproj file has a package reference to Microsoft.AspNetCore.All, as shown in the following listing.

Listing 2.3 hwwebapp.csproj package reference to Microsoft.AspNetCore.All

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference
      Include="Microsoft.AspNetCore.All"
      Version="2.0.0" />
  </ItemGroup>

</Project>
```

You won't use this folder in this example.

PackageReference references a NuGet package.

2.4.1 ASP.NET Core uses the Kestrel web server

Web applications need a web server, and Kestrel is the web server that was built for ASP.NET Core. It can be started from a console application and is included as part of the Microsoft.AspNetCore.All metapackage (a package that references a bunch of other packages). In the following listing, Kestrel is included as part of the default `WebHost` builder.

Listing 2.4 Program.cs for an ASP.NET Core web application

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
```

The usings are trimmed to only what's needed.

```
namespace hwwebapp
{
  public class Program
  {
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
    }
  }
}
```

Starts the Kestrel web server

References the Startup class in Startup.cs

2.4.2 Using a Startup class to initialize the web server

Next, let's look at the `Startup` class referenced from Program.cs. This class is used by ASP.NET to define how requests will be handled. Your web service will simply return "Hello World" in the response.

The Startup.cs file included with the template includes more than is necessary. The following listing shows a trimmed-down version of this class.

Listing 2.5 Trimmed-down Startup.cs file for a Hello World web application

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;

namespace hwwebapp
{
  public class Startup
  {
    public void Configure(IApplicationBuilder app,
      IHostingEnvironment env, ILoggerFactory loggerFactory)
    {
      loggerFactory.AddConsole();

      if (env.IsDevelopment())
      {
        app.UseDeveloperExceptionPage();
      }

      app.Run(async (context) =>
        await context.Response.WriteAsync(
          "Hello World")
      );
    }
  }
}
```

These are created by dependency injection, covered in chapter 6.

Log messages go to the console.

Exceptions are hidden from web pages except in development mode.

Responds to all requests with "Hello World"

Whereas the `Program` class starts the web server, the `Startup` class starts the web application.

There's a lot of stuff to unpack in listing 2.5. This book doesn't delve deeply into ASP.NET Core, but anonymous methods, async/await, dependency injection, and logging are all covered in later chapters.

For those not familiar with C#

If you aren't familiar with C#, the => may be confusing. This is used to create an anonymous method—*anonymous* meaning that it has no name. The arguments for the method go on the left side of the =>, which is the `HttpContext` in listing 2.5. The method definition goes on the right side. If the method needs only one line and returns a value, you can forgo the brackets, `return`, and `;` and keep only the expression.

In this case, you want to write "Hello World" in the HTTP response and return the asynchronous `Task` object. We'll cover the subject of tasks later in the book.

Pros and cons of anonymous methods

Anonymous methods in C# can provide a huge productivity boost. In addition to making code more readable, there's another often-overlooked benefit: you don't have to figure out a name for a method. I'm surprised by how much time I spend thinking about how to name various artifacts in my code.

Some of the drawbacks to anonymous methods are evident in debugging. Anonymous methods still get names in stack traces, just not recognizable ones. It's also difficult to manually create breakpoints on or in anonymous methods from a debugger when you don't know the method name.

2.4.3 Running the Hello World web application

To run the web application, execute the `dotnet run` command at the command line, just as before. This starts the web server, which should produce output like the following:

```
Hosting environment: Production
Content root path: /hwwebapp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

The web address for the app

The output of `dotnet run` includes the web address for the Kestrel server. In this example it's http://localhost:5000. Open a browser and navigate to this address. Figure 2.1 shows the Hello World web application running in the browser.

Figure 2.1 Hello World web application in the browser

2.5 Creating an ASP.NET Core website from the template

In the previous section, you created a minimal ASP.NET Core web application. That may be a useful starting point for creating a service, but a website requires a lot more work. The .NET Core SDK includes a template for ASP.NET Core websites to help get you started. If you're familiar with ASP.NET development in Visual Studio, this is closer to the New Project scenario.

To use the template, start from a new folder called "hwwebsite," and in that folder, execute this command: `dotnet new mvc`. List the contents of the folder, and you'll find a lot more files than you created in the previous template.

Run the `dotnet run` command to get the web server started as before. Then visit http://localhost:5000 in your browser to see the default website.

2.6 Deploying to a server

Now that you've written some applications, you can try deploying them to a server. .NET Core was designed to deploy by simply copying files. It helps to know where these files come from.

In figure 2.2, the .NET CLI package contains some assemblies that are installed into the dotnet folder. They're enough to power the .NET CLI (command-line interface) but not much else. When the SDK does the restore, it determines what packages it needs to download based on your dependencies. Kestrel is an example of a dependency that isn't part of .NET Core but is necessary to run an ASP.NET application.

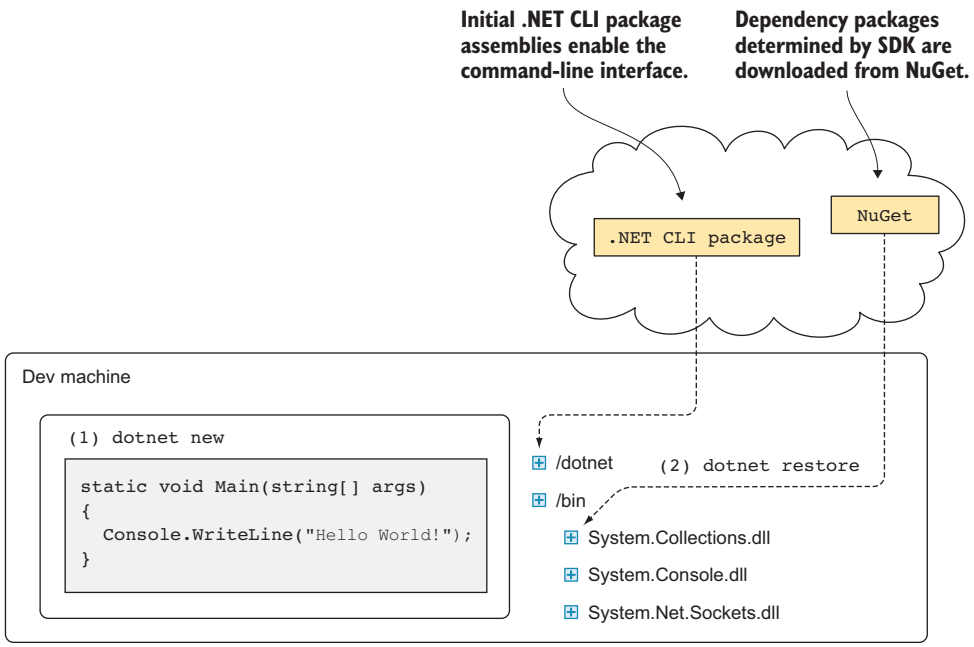


Figure 2.2 Locations of components and assembly files for .NET Core

The .NET CLI downloads these dependency packages from a package store called NuGet and stores them centrally, so you don't have to download them for every project you write.

2.6.1 Publishing an application

The .NET CLI provides a way to consolidate all the binaries needed to deploy your application into one folder. From the folder containing the Hello World console application (hwapp), execute `dotnet publish -c Release` from the command line or terminal. The console output indicates to which folder the binaries are published.

Change to the publish folder under the output folder. There should be four files:

- hwapp.deps.json
- hwapp.dll
- hwapp.pdb
- hwapp.runtimeconfig.json

Copy these files to another machine that has the .NET Core SDK installed. Then, to execute the application, run the command `dotnet hwapp.dll` from the folder containing the copied files.

In this example, only the binaries built for the project, and any included packages, are published. This is called a *framework-dependent deployment*. When you deploy the files from the publish folder, you need to have the .NET Core SDK installed on the target machine.

PUBLISHING A SELF-CONTAINED APPLICATION

There's another way to create an application that will include all of the .NET Core assemblies, so it can run on a machine that doesn't have the .NET Core SDK installed. This is called a *self-contained application*.

To create a self-contained application, you'll first need to be explicit about what runtimes your application can run on. This is done by adding the `RuntimeIdentifiers` property to your project file, as shown in the following listing.

Listing 2.6 Add `RuntimeIdentifiers` to the hwapp.csproj file

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <RuntimeIdentifiers>win10-x64;osx.10.11-x64;linuxmint.17.1-x64
    </RuntimeIdentifiers>
  </PropertyGroup>
</Project>
```

Now you can create a self-contained application by using the runtime identifier in the publish command, as follows:

```
dotnet publish -c Release -r linuxmint.17.1-x64
```

Listing the contents of the bin\Release\netcoreapp2.0\linuxmint.17.1-x64\publish folder will reveal a lot of files. Figure 2.3 illustrates what this publishing step looks like.

Packages are stored either in the local .NET CLI cache or in a NuGet package library. The restore process pulls all the files to the cache. The `dotnet publish` command with the runtime option collects all the files necessary, including the binaries built from your project code, and puts them into the publish folder. The full contents of this folder can then be copied to other machines or containers.

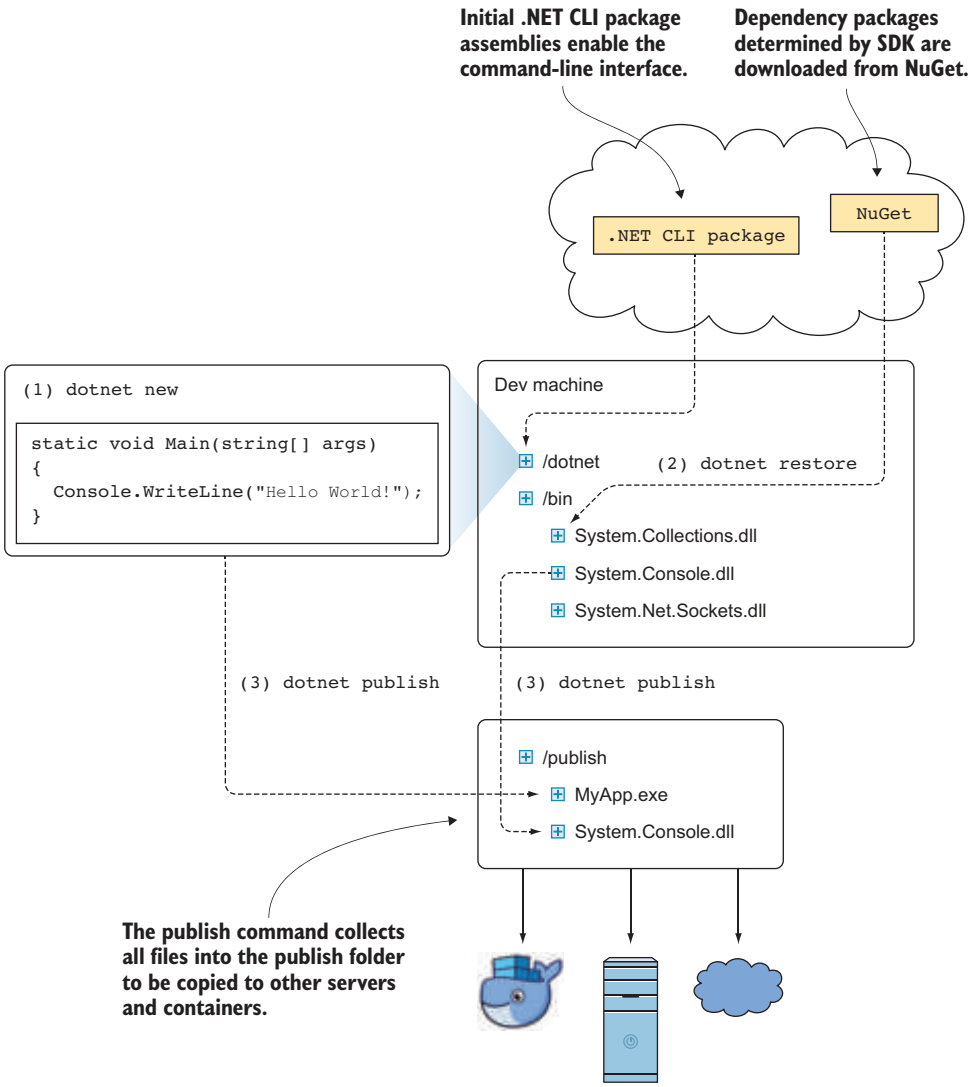


Figure 2.3 How files used by .NET Core applications are published

2.6.2 Deploying to a Docker container

Deploying to a Linux, macOS, or Windows machine is all the same—copy the contents of the publish folder to a folder on the target machine. If you're not deploying a self-contained application, the only prerequisite for the target machine is having the .NET Core SDK installed. For Docker, the same general theory applies, but you'll typically want to create a container with your application.

Earlier, you used Microsoft's .NET Core container by executing the following command from the Docker CLI:

```
docker run -it microsoft/dotnet:latest
```

You get the latest .NET Core container from Docker Hub by using `microsoft/dotnet:latest`.

To create a new Docker container with the application, you'll start with the .NET Core container, copy your published application's files, and tell it to run the application.

First, open a command prompt with the Docker CLI. Change to the folder containing the Hello World console application, and then create a new text file in the current folder called Dockerfile. Insert the text from the following listing.

Listing 2.7 Dockerfile for Hello World console application

```
FROM microsoft/dotnet:latest
COPY bin/Release/netcoreapp2.0/publish/ /root/
ENTRYPOINT dotnet /root/hwapp.dll
```

Save and close the Dockerfile file. Then execute the following Docker CLI command to build the container.

Listing 2.8 Command to build Hello World console application container image

```
docker build -t hwapp .
```

Now you can run the container with the following command and see that your application is working.

Listing 2.9 Running the Hello World console application Docker container

```
$ docker run -it hwapp
Hello World
```

Deploying the web application

The barebones ASP.NET Core web application you created earlier (hwwebapp) will need some adjustments before it can work on Docker. First, modify the Program.cs file as shown in the following listing.

Listing 2.10 Modifying Program.cs for Docker container deployment

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace hwwebapp
{
  public class Program
  {
    public static void Main(string[] args)
    {
      BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
        .UseUrls("http://*:5000/")
        .UseStartup<Startup>()
        .Build();
  }
}
```

Add this line.

The added line allows URLs other than http://localhost:5000. Because the application is being deployed to a container, you'll need to test it by pinging it from outside the container.

You also need to configure Docker to open the port. To do this, you'll first need to edit the Dockerfile, as shown in the following listing

Listing 2.11 Modifying Dockerfile for Docker container deployment

```
FROM microsoft/dotnet:latest
COPY bin/Release/netcoreapp2.0/publish/ /root/
EXPOSE 5000/tcp
ENTRYPOINT dotnet /root/hwwebapp.dll
```

Opens port 5000

Build and run the application and Docker container with the following commands:

```
dotnet publish -c Release
docker build -t hwwebapp .
docker run -it -p 5000:5000 hwwebapp
```

To test the web application, use localhost with port 5000 in your browser: http://localhost:5000.

2.6.3 Packaging for distribution

.NET Core makes it easy to package your library or application so that others can use it in their applications. This is accomplished through the `dotnet pack` command.

Try this out on the Hello World console application. Go to the hwapp folder and run `dotnet pack -c Release`. This command will create a hwapp.1.0.0.nupkg file.

The .nupkg file produced by the `dotnet pack` command is a NuGet package. A lot of the dependencies you use in your .NET Core applications will come from NuGet packages. They're a great way of sharing code both privately and publicly. We'll cover these in more depth in chapter 12.

2.7 Development tools available for .NET Core

Microsoft typically released new versions of the .NET Framework in tandem with updates to Visual Studio. This meant that the files involved in .NET projects didn't need to be human-readable, because the IDE would take care of that work. When Microsoft started building .NET Core, they needed simple files that developers could edit directly with text editors. They relied on JSON and created a custom build system.

In the long term, maintaining two different build systems would be costly and confusing, so even though developers loved the new JSON-based .NET Core build system, the Microsoft team needed to incorporate it back into their existing build system, called MSBuild. The chief complaint against MSBuild was about the complexity of the project files. Luckily, MSBuild is flexible enough that the .NET Core team could address this concern.

The .NET Core project files use a lot of techniques to reduce the file size and complexity so that developers can more easily edit them by hand. This means you can use any text editor you want for writing your .NET Core applications, instead of needing to use Visual Studio. The range of development tools includes everything from full-featured IDEs to vi, notepad, or butterflies (see https://xkcd.com/378). For this book, you'll only need a basic text editor and a command prompt. But you can also try out some of these editors to improve your experience.

2.7.1 OmniSharp

OmniSharp is a family of open source projects whose goal is to enable building .NET in the most popular text editors. It does this with tools, editor integrations, and libraries. OmniSharp provides plugins for all kinds of text editors:

- Atom (atom.io)
- Sublime (sublimetext.com)
- Brackets (brackets.io)
- Emacs (gnu.org/software/emacs)
- Vim (vim.org)
- Visual Studio Code (code.visualstudio.com)

As an example, install the OmniSharp extension for Visual Studio Code.

OMNISHARP FOR VISUAL STUDIO CODE

Visit https://code.visualstudio.com from your browser and download Visual Studio Code. After installing it, you can run the command `code .` from a new terminal or command line window to open Visual Studio Code on the current folder.

Mac users need to take an extra step: open the Command Palette from the View menu, and type in "shell command" to find "Install 'code' command in PATH".

When you open a C# file with VS Code for the first time, it will prompt you to install an extension. The one entitled "C#" is the OmniSharp extension. This extension should be all you need to build, run, and debug .NET Core applications—assuming you've already installed the .NET SDK.

Also note that VS Code has an integrated terminal that can be configured to Windows Command Prompt, PowerShell, or Bash. You can also have multiple terminals open at the same time. This makes it easier to try out the various .NET CLI commands used throughout this book.

2.7.2 Visual Studio for Mac

Developers who may have worked with Xamarin Studio in the past will recognize Visual Studio for Mac as being Xamarin Studio rebranded. VS for Mac has several editions, including a free community edition.

> **BUILDING XAMARIN APPS ON WINDOWS** If you want the Xamarin Studio experience but aren't on Mac, Xamarin is an option you can choose from the Visual Studio 2017 installer.

Creating new projects in Visual Studio for Mac is done through the New Project wizard (shown in figure 2.4). You can use this instead of the `dotnet` command-line options. Also note that VS for Mac will perform restores for you.

2.7.3 Visual Studio 2017

Visual Studio has always been the flagship development environment for Microsoft. It only runs on Windows and has been synonymous with Windows and the .NET Framework for many years. Only in recent releases has Visual Studio started to support other languages, such as Python and JavaScript. You can use the preview .NET Core tooling in Visual Studio 2015, but it doesn't come with support. Visual Studio 2017 has a free Community Edition that includes .NET Core and Xamarin tooling. You can find it at www.visualstudio.com/downloads.

With Visual Studio, you don't need to use the `dotnet` command-line commands. For example, the New Project wizard shown in figure 2.5 replaces the `dotnet new` functionality.

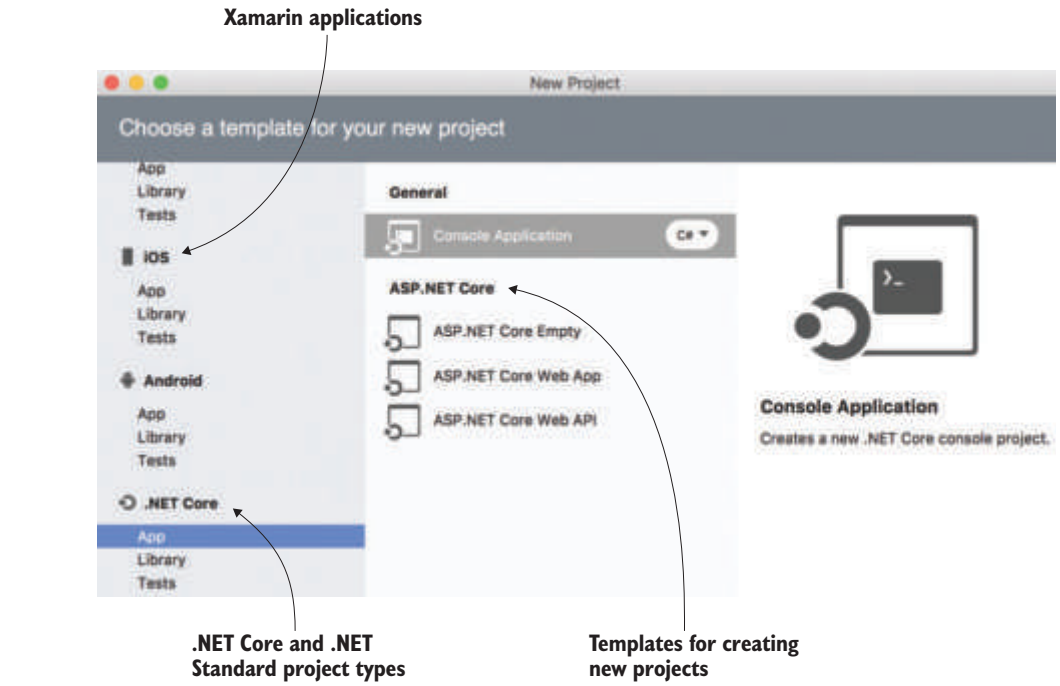


Figure 2.4 Visual Studio for Mac's New Project wizard

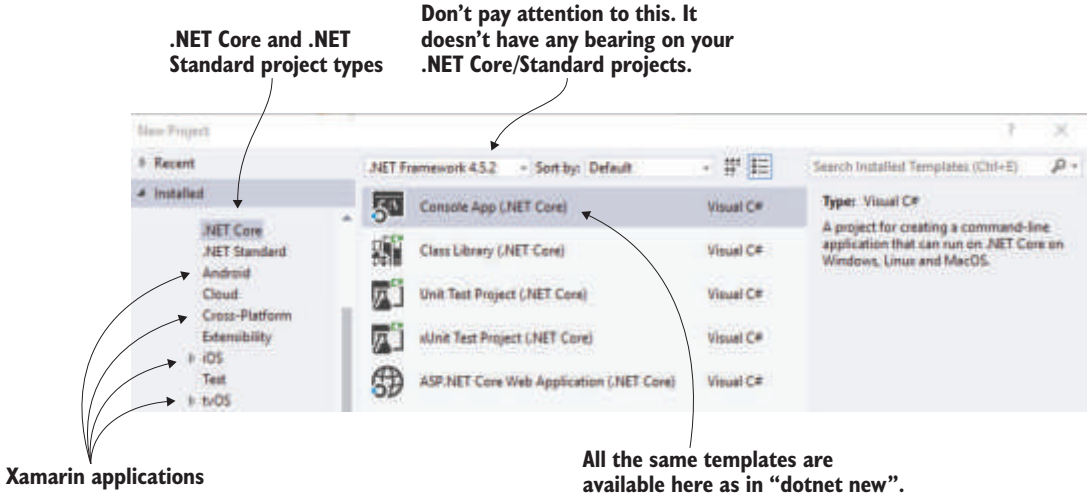


Figure 2.5 Visual Studio 2017 Community edition New Project wizard

Figure 2.6 Right-click menu for a .NET Core project in Visual Studio 2017

Right-click on the project in the Solution Explorer to find some of the other commonly used .NET CLI commands. Figure 2.6 shows what's available.

Note that `dotnet restore` isn't mentioned anywhere. That's because Visual Studio will automatically detect when a restore is needed. There's no explicit Run command, but you can use the traditional F5 or Ctrl-F5, as with any other Visual Studio project.

Additional resources

I introduced a lot of concepts in this chapter. Here are some helpful links where you can find out more information about the subjects and products we touched on:

- The central .NET site—https://dot.net
- Install instructions for .NET Core—www.microsoft.com/net/core
- Docker—www.docker.com
- *Docker in Action* by Jeff Nickoloff—http://mng.bz/JZSJ
- *Docker in Practice, Second Edition*, by Ian Miell and Aidan Hobson Sayers—http://mng.bz/H42I
- *ASP.NET Core in Action* by Andrew Lock—http://mng.bz/DI1O
- Node.js—https://nodejs.org
- *Node.js in Action, Second Edition*, by Alex Young, Bradley Meck, and Mike Cantelon—http://mng.bz/mK7C
- Bower package manager—https://bower.io

Summary

In this chapter you learned how to get started with .NET Core by doing the following:

- Installing the .NET Core SDK
- Learning basic .NET CLI commands
- Building both a console application and an ASP.NET Core web application
- Preparing applications for deployment and distribution

These are all fundamental skills for developing with .NET Core. Here are a few tips to keep in mind:

- The `dotnet new` command makes it easy to get a template for starting a new project. It's also nice if you need a quick reference.
- Whenever you make changes to the dependencies, you can run the `dotnet restore` command to get the packages immediately.
- Use the Yeoman generator for more customizable templates.

You'll practice more with the .NET CLI commands (`restore`, `new`, `run`, `publish`, and `pack`) throughout the book. In the next chapter, we'll get into the details of the project files and you'll learn how to build more substantial applications with .NET Core.

11 Multiple frameworks and runtimes

This chapter covers

- The .NET Portability Analyzer
- Building projects that work on multiple frameworks
- Handling code that's operating-system specific

There are two features of .NET Core that we'll look at in this chapter. One is the ability to run .NET Core applications on many different operating systems. The other is the ability to write .NET code specific to each .NET framework if you need the code to operate differently.

You can take advantage of these capabilities in your own applications and libraries, which is particularly useful when you have to extend beyond the .NET Standard. It's also useful when you're trying to use OS-specific features or native components as the interfaces, because these will be different on each OS.

11.1 Why does the .NET Core SDK support multiple frameworks and runtimes?

The .NET Core SDK supports building for multiple frameworks. You can specify the desired framework with a command-line option.

Consider these examples:

```
dotnet build --framework netcoreapp2.0
dotnet run --framework netcoreapp2.0
dotnet test --framework netcoreapp2.0
```

So far in this book we've only targeted one framework at a time—either `netstandardxxx` or `netcoreappxxx`—so there was no occasion to exercise this capability.

If you're building a new library that adheres to the .NET Standard, it will work universally with other .NET frameworks. If you're porting a library from either Xamarin or the .NET Framework, it may be able to port directly to the .NET Standard Library without modifying the code. There are cases, though, where your code needs to be built for multiple frameworks.

For instance, suppose you have code that uses XAML that you want to make work on the .NET Framework, Xamarin Forms, and Universal Windows Applications. Or maybe your library is used by some existing applications that you can't change. The .NET Core SDK makes it possible to support multiple frameworks in the same NuGet package (generated by `dotnet pack`).

> **FRAMEWORKS VS. RUNTIMES** Runtimes and frameworks are not the same thing. A *framework* is a set of available APIs. A *runtime* is akin to an operating system (see section 3.1.3 for more details). Your code may have to work with some OS-specific APIs, which means that it will have different code for different runtimes.

One example of a library that works differently depending on the runtime is the Kestrel engine, which is used for hosting ASP.NET Core applications. Kestrel is built on a native code library called libuv. Because libuv works on multiple operating systems, it's a great foundation for the flagship ASP.NET Core web server. But even libuv has its limitations, so Kestrel doesn't work on all platforms.

Another example of needing to support multiple runtimes is the System.IO.Compression library. Instead of implementing Deflate/GZip compression in .NET managed code, System.IO.Compression relies on a native library called zlib. The zlib library isn't only the de facto standard for GZip compression and decompression, it's also implemented in native code, which gives it a slight performance advantage over any managed .NET implementation. Because zlib is a native library, the code in System.IO.Compression has to behave differently based on the runtime.

The .NET Standard Library gives you a great foundation on which to build libraries and applications for a broad array of platforms, but it's not comprehensive. Luckily, the .NET Core SDK is flexible enough to support different frameworks and runtimes, which can allow you to consolidate code into a single project and simplify packaging and distribution. This chapter introduces some techniques for supporting multiple frameworks and runtimes.

You'll start by trying to port code between .NET frameworks.

11.2 .NET Portability Analyzer

The .NET Portability Analyzer helps you migrate from one .NET framework to another. See figure 11.1, which shows that Xamarin, .NET Core, and the .NET Framework are all frameworks that implement the .NET Standard. The .NET Portability Analyzer has detailed information on where each framework deviates from the standard and how that translates into other frameworks.

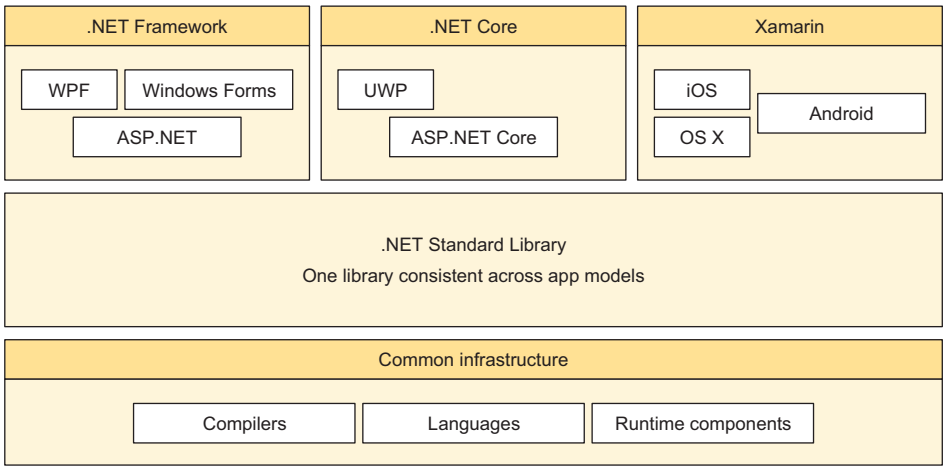


Figure 11.1 .NET Framework, .NET Core, and Xamarin are all different frameworks that support the .NET Standard Library.

If you want to port your Xamarin or .NET Framework library to .NET Core, the .NET Portability Analyzer can help. It identifies all the incompatibilities between the two frameworks and provides suggestions, where possible. The tool is available both as a command-line executable and a Visual Studio plugin. We'll explore the Visual Studio plugin version.

11.2.1 Installing and configuring the Visual Studio 2017 plugin

In Visual Studio, open the Tools menu and choose Extensions and Updates. In the Extensions and Updates dialog box, pick Online in the tree in the left pane. Type "portability" in the search box, and look for the .NET Portability Analyzer (shown in figure 11.2).

Download and install the plugin. After installing, you'll need to restart Visual Studio.

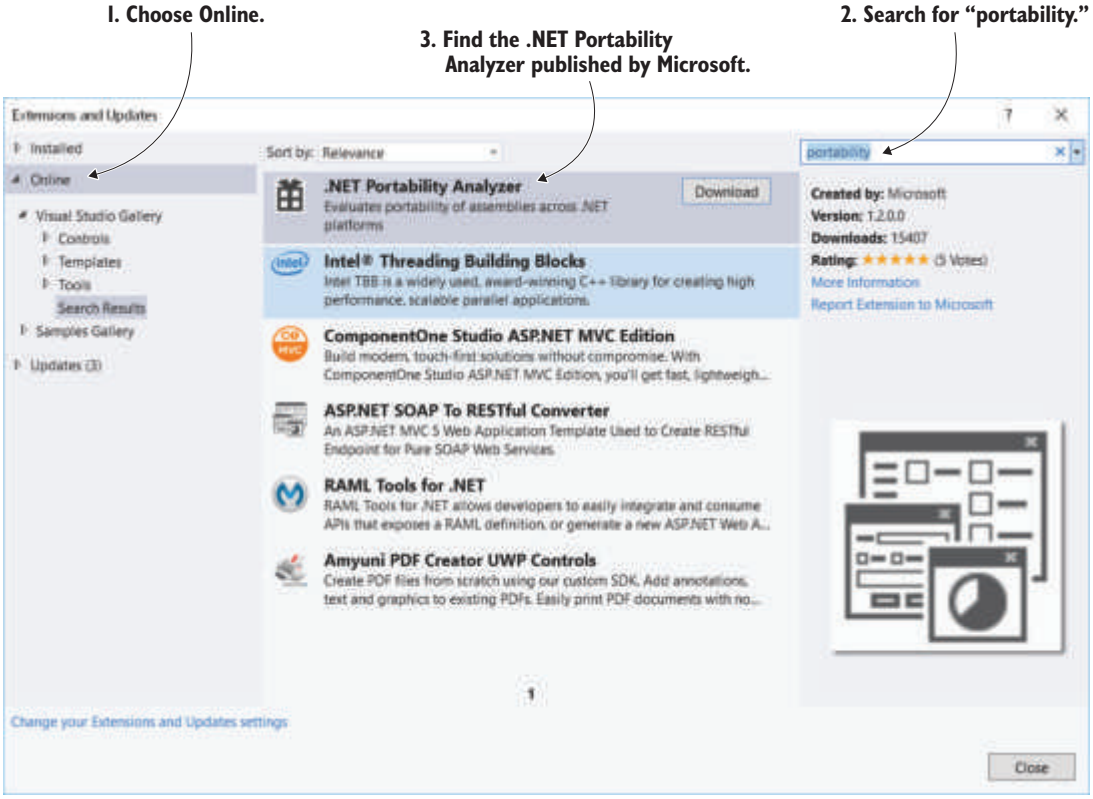


Figure 11.2 Search for the .NET Portability Analyzer.

11.2.2 Sample .NET Framework project

Now create a new project to test out the .NET Portability Analyzer. The sample project will execute a simple latency test by making HTTP requests to a given URI.

Create a new C# console application (listed as Console App (.NET Framework) in the New Project dialog box) in Visual Studio targeting .NET Framework version 4.5 or later. Alter the Program.cs file to contain the following code.

Listing 11.1 Program.cs for your test of the .NET Portability Analyzer

```
using System;
using System.Diagnostics;
using System.IO;
using System.Net;

namespace ConsoleApplication1
{
  class Program
  {
```

```
    static void Main(string[] args)
    {
      string uri = "http://www.bing.com";
      var firstRequest = MeasureRequest(uri);
      var secondRequest = MeasureRequest(uri);
      if (firstRequest.Item1 != HttpStatusCode.OK &&
          secondRequest.Item1 != HttpStatusCode.OK) {
        Console.WriteLine("Unexpected status code");
      } else {
        Console.WriteLine($"First request took {firstRequest.Item2}ms");
        Console.WriteLine($"Second request took {secondRequest.Item2}ms");
      }
      Console.ReadLine();
    }

    static Tuple<HttpStatusCode, long> MeasureRequest(string uri)
    {
      var stopwatch = new Stopwatch();
      var request = WebRequest.Create(uri);
      request.Method = "GET";
      stopwatch.Start();
      using (var response = request.GetResponse()
             as HttpWebResponse)
      {
        using (var reader = new StreamReader(response.GetResponseStream()))
        {
          reader.ReadToEnd();
          stopwatch.Stop();
        }

        return new Tuple<HttpStatusCode, long>(
            response.StatusCode,
            stopwatch.ElapsedMilliseconds);
      }
    }
  }
}
```

MeasureRequest measures the latency of an HTTP request.

Makes sure you've read the whole response

The preceding code is a contrived example that measures the latency of web requests. It creates a `WebRequest` object pointing to the URI passed in. The response object exposes the `GetResponseStream` method, because the response may be large and take some time to download. Calling `ReadToEnd` makes sure you get the full content of the response.

The first request from the example project takes longer for many reasons, such as JIT compiling and setting up the HTTP connection. The latency for the second request is a more realistic measurement of the time it takes to get a response from the endpoint (http://www.bing.com in this case).

11.2.3 Running the Portability Analyzer in Visual Studio

Let's see how this code would port to .NET Core. First, change the settings for the Portability Analyzer. Open the settings as shown in figure 11.3. Choose all the options for .NET Core target platforms, as shown in figure 11.4.

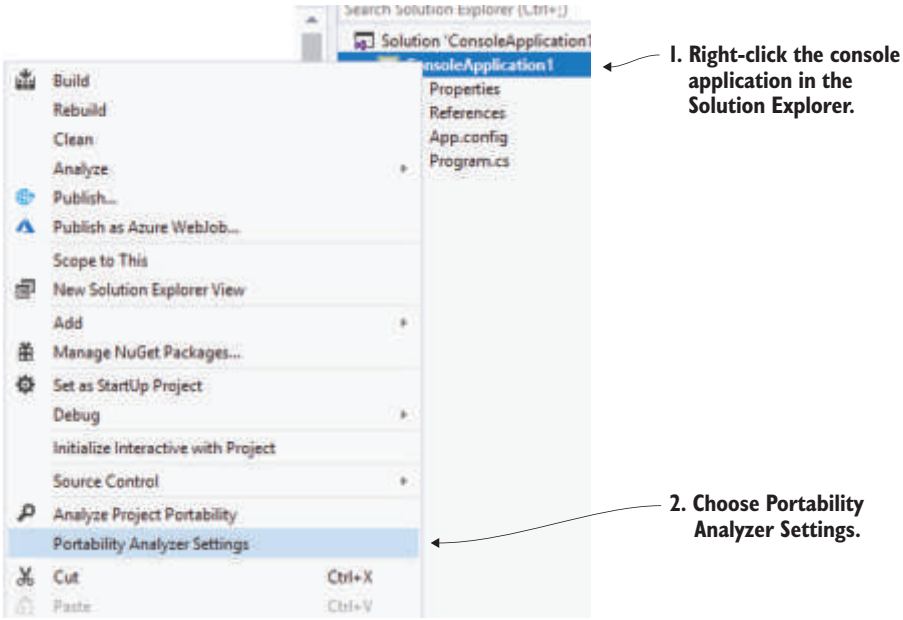


Figure 11.3 Open the settings for the Portability Analyzer.

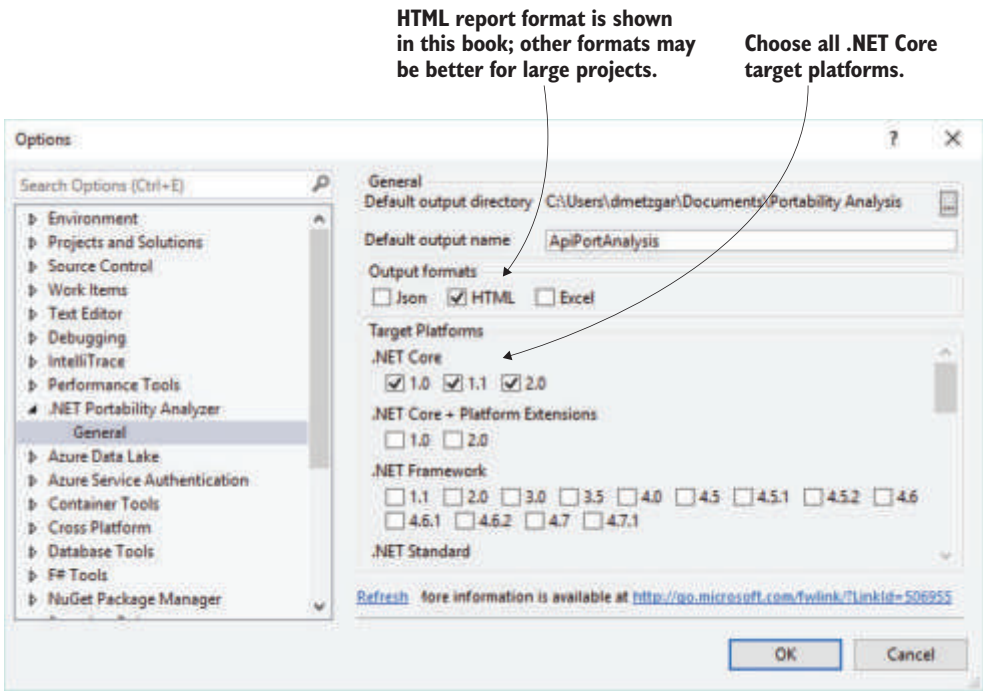


Figure 11.4 Choose all .NET Core Target Platforms in the Portability Analyzer settings.

Now run the Portability Analyzer on your project. This option is also in the project's right-click menu, shown in figure 11.5.

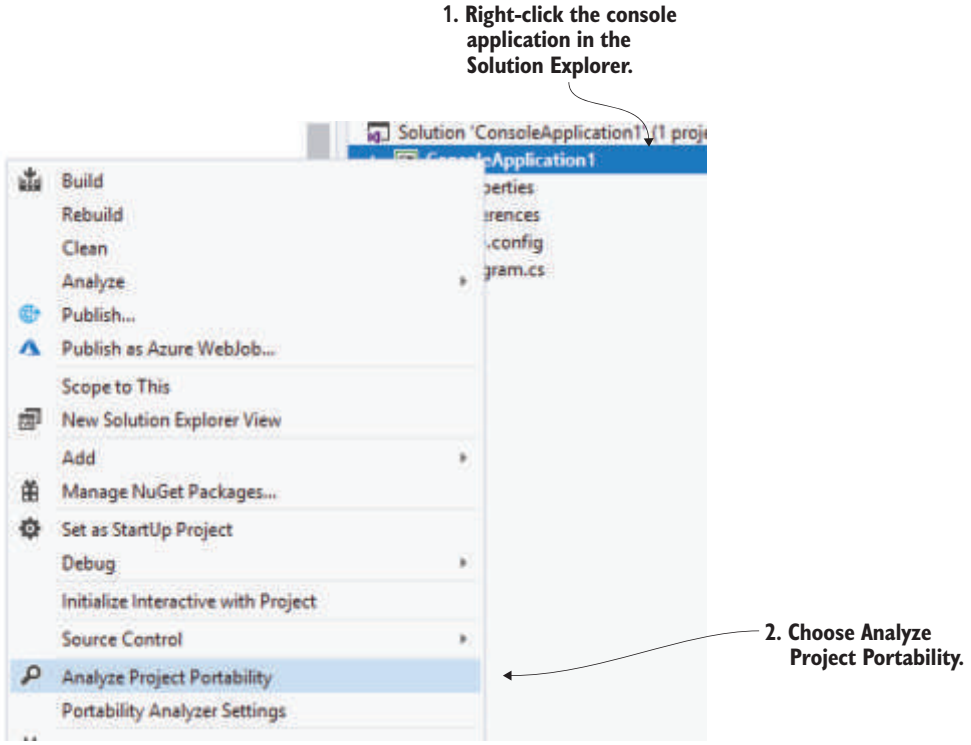


Figure 11.5 Run the Portability Analyzer from the right-click menu.

After the analyzer is finished, the Portability Analyzer Results pane will pop up, as shown in figure 11.6.

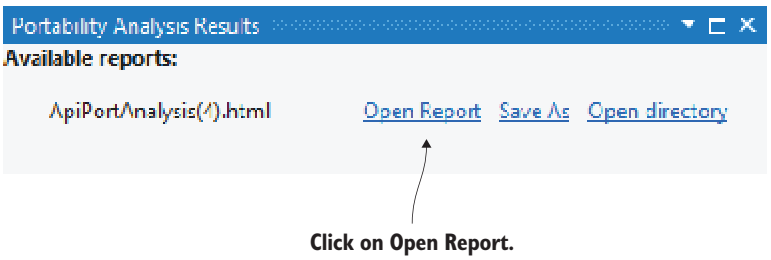


Figure 11.6 Portability Analyzer Results pane

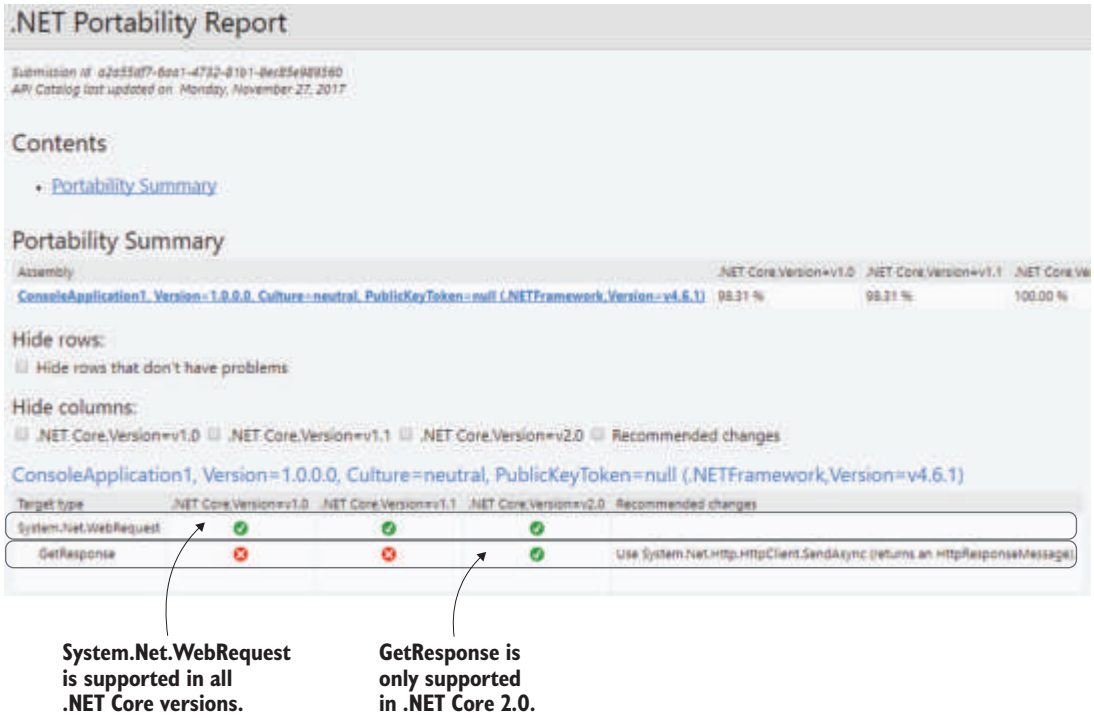


Figure 11.7 Portability analysis of the sample code

Figure 11.7 shows an HTML version of the report.

If you're targeting .NET Core 1.0 or 1.1, the suggested method for fixing the code is to use a different means of making HTTP requests entirely, via `HttpClient`. You learned about `HttpClient` back in chapter 7. Change Program.cs to use `HttpClient` as shown in the following listing.

Listing 11.2 New method that implements the suggestion from the Portability Analyzer

```
using System.Net.Http;                                  <- Add this using statement.

class Program
{
  static HttpClient client = new HttpClient();

  static Tuple<HttpStatusCode, long> MeasureRequest(string uri)
  {
    var stopwatch = new Stopwatch();
    stopwatch.Start();
    var response = client.GetAsync(uri).Result;         <- Result waits for the Task to finish and gets the result.
    response.Content.ReadAsStringAsync().Wait();        <- You don't need the result here, so you just Wait().
    stopwatch.Stop();
```

```
    return new Tuple<HttpStatusCode, long>(
      response.StatusCode,
      stopwatch.ElapsedMilliseconds);
  }
}
```

Run the Portability Analyzer again and you'll see you're now at 100%.

In this case there was a suitable substitute that also works in the .NET Framework. In the next section, we'll look at how to handle cases where there isn't a substitute that works in both frameworks.

11.3 Supporting multiple frameworks

In the previous example, you were able to replace the old .NET Framework code with its .NET Standard equivalent. But this may not always be possible.

Consider the following code, written for the .NET Framework.

Listing 11.3 `EventProvider` .NET Framework sample

```
using System;
using System.Diagnostics.Eventing;

namespace ConsoleApplication3
{
  class Program
  {
    private static readonly Guid Provider =
      Guid.Parse("B695E411-F53B-4C72-9F81-2926B2EA233A");

    static void Main(string[] args)
    {
      var eventProvider = new EventProvider(Provider);
      eventProvider.WriteMessageEvent("Program started");

      // Do some work

      eventProvider.WriteMessageEvent("Program completed");
      eventProvider.Dispose();
    }
  }
}
```

The actual Guid is not important.

Writes events to Windows

You may have legacy code that uses some Windows-specific features like the preceding code. This code produces an event in Windows under a given provider Guid. There may be logging tools that listen for these events, and slight changes in how the events are emitted might break those tools.

11.3.1 Using EventSource to replace EventProvider

Try running the .NET Portability Analyzer on the preceding code to see the suggested .NET Core alternative. Figure 11.8 shows an example analysis.

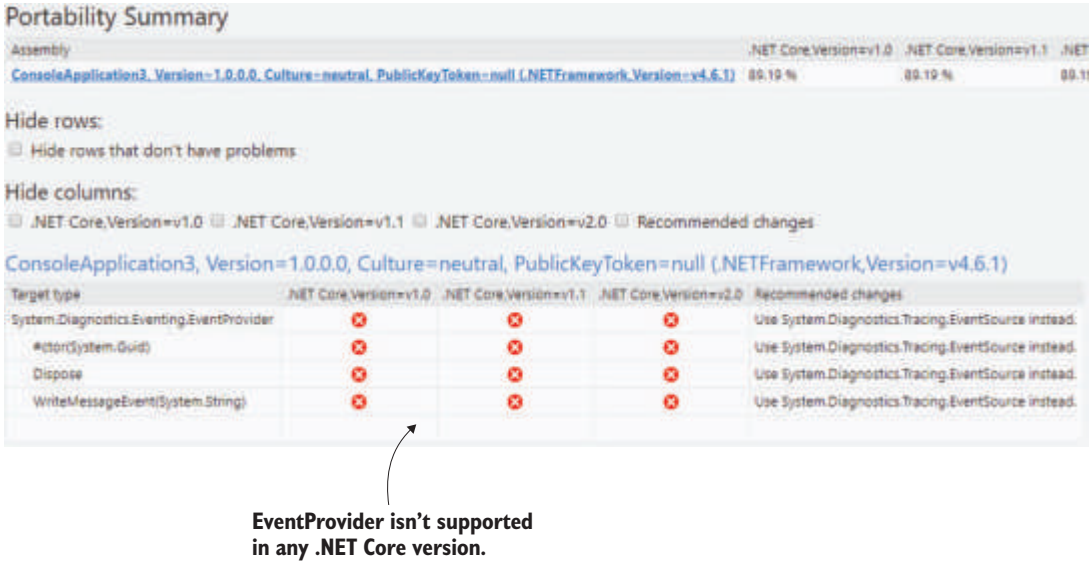


Figure 11.8 Portability analysis of the sample code using `EventProvider`

The recommended change in this case is to use an `EventSource`. An `EventSource` is definitely the way to go when writing events without relying on platform-specific features. You learned about `EventSource` back in chapter 10. Unfortunately, if you're replacing an existing Windows event provider, the `EventSource` implementation may not produce the exact same events.

Let's look at a similar version written for .NET Core using `EventSource`, shown in the following listing.

Listing 11.4 Writes events using `EventSource`

```
using System.Diagnostics.Tracing;

namespace SampleEventSource
{
  [EventSource(Name = "My Event Source",
   Guid = "B695E411-F53B-4C72-9F81-2926B2EA233A")]
  public sealed class MyEventSource : EventSource
  {
    public static MyEventSource Instance =
      new MyEventSource();

    [Event(1,
      Level = EventLevel.Informational,
      Channel = EventChannel.Operational,
      Opcode = EventOpcode.Start,
      Task = Tasks.Program,
```

Same provider Guid

Helper singleton instance

EventSource allows more customization of events.

Tasks are required when using an Opcode.

```
      Message = "Program started")]
    public void ProgramStart()
    {
        WriteEvent(1);
    }

    [Event(2,
      Level = EventLevel.Informational,
      Channel = EventChannel.Operational,
      Opcode = EventOpcode.Stop,
      Task = Tasks.Program,
      Message = "Program completed")]
    public void ProgramStop()
    {
        WriteEvent(2);
    }

    public class Tasks
    {
      public const EventTask Program = (EventTask)1;
    }
  }
}
```

Start and Stop are standard Opcodes.

In the preceding code, you took advantage of some of the capabilities that `EventSource` has to offer. It also makes the `Program` code much cleaner, as you can see in the following listing.

Listing 11.5 Program.cs refactored to use the new `EventSource`

```
using System;

namespace SampleEventSource
{
  public class Program
  {
    public static void Main(string[] args)
    {
      MyEventSource.Instance.ProgramStart();

      // Do some work

      MyEventSource.Instance.ProgramStop();
    }
  }
}
```

The events are slightly different than before, so there's a risk that the new code will break existing tools. But because those tools will have to be changed to work with the .NET Core version of the application anyways, don't worry about making the events exactly the same. Instead, you'll focus on allowing the .NET Framework version of the application to continue to work as before. That means you have to support multiple frameworks.

Start by creating the .NET Core project. Create a folder called SampleEventSource, and open a command prompt in that folder. Run `dotnet new console` to create a new .NET Core console application. Modify the Program.cs file to match listing 11.5. Also create a new file called MyEventSource.cs with the code in listing 11.4.

Feel free to build and run the application. You won't see any output from it. To view the logs that are emitted from the `EventSource`, you'll need to create a consumer, which was covered in chapter 10. For this chapter, we'll just assume it works.

11.3.2 Adding another framework to the project

Indicating support for another framework is straightforward. Modify the SampleEventSource.csproj file as follows.

Listing 11.6 csproj for sample with .NET Framework support

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>

    <TargetFrameworks>netcoreapp2.0;net46</TargetFrameworks>
    <RuntimeFrameworkVersion
      Condition=" '$(TargetFramework)' == 'netcoreapp2.0' "
      >2.0.0-*</RuntimeFrameworkVersion>
  </PropertyGroup>

  <ItemGroup Condition=" '$(TargetFramework)' == 'net46' ">
    <Reference Include="System" />
    <Reference Include="Microsoft.CSharp" />
  </ItemGroup>

</Project>
```

Change TargetFramework to TargetFrameworks.

RuntimeFramework-Version is only set for netcoreapp2.0.

References are only needed for building with .NET Framework.

Notice that you specifically need `net46`. `net45` won't work in this case because the `EventChannel` class wasn't defined in that version. If you remove the `Channel` specification from the `MyEventSource` class, however, you should be able to use`net45`.

You can now build this code for the .NET Framework using the following command:

```
dotnet build --framework net46
```

This will use the `EventSource` on the .NET Framework and .NET Core, but you want it to revert to the old code when using the .NET Framework. Because the framework is something you know at build time, you can use preprocessor directives. Listing 11.7 shows how this works.

WHAT IS A PREPROCESSOR DIRECTIVE? A preprocessor directive is a statement that's executed before compilation starts. If you're familiar with C or C++, you may be familiar with creating macros using preprocessor directives. Although macros aren't available in C#, you can still have conditionally compiled code.

Listing 11.7 Program.cs rewritten to use preprocessor directives

```
using System;

#if NET46
using System.Diagnostics.Eventing;
#endif

namespace SampleEventSource
{
  public class Program
  {
#if NET46
    private static readonly Guid Provider =
      Guid.Parse("B695E411-F53B-4C72-9F81-2926B2EA233A");
#endif

    public static void Main(string[] args)
    {
#if NET46
      var eventProvider = new EventProvider(Provider);
      eventProvider.WriteMessageEvent("Program started");
#else
      MyEventSource.Instance.ProgramStart();
#endif

      // Do some work

#if NET46
      eventProvider.WriteMessageEvent("Program completed");
      eventProvider.Dispose();
#else
      MyEventSource.Instance.ProgramStop();
#endif
    }
  }
}
```

NET46 is automatically defined.

The `#if` and `#endif` are preprocessor directives that will include the code contained between them only if `NET46` is defined. `NET46` is created automatically from the name of the framework. The special characters are usually replaced with underscores, and everything is in uppercase. For instance, the framework moniker `netcoreapp2.0` would be defined as `NETCOREAPP2_0`.

ALTERNATIVES TO PUTTING #IF/#ENDIF IN THE MIDDLE OF YOUR CODE Putting `#if` directives all over your code can make it hard to read. There are a couple of ways that I avoid this. The first is to have two copies of the file (for example, one for `NET46` and one for `NETCOREAPP2_0`) with `#if/#endif` surrounding the entire contents of each file. Another way is to also have these two different versions of the file, but to exclude or include one based on conditions in the project file. This has the obvious drawback of

maintaining two files, so it's helpful to isolate the framework-specific code in one class to reduce duplication.

You should now be able to build the application by specifying the target moniker at the command line, as follows:

```
dotnet build --framework net46
dotnet build --framework netcoreapp2.0
```

You can also build all frameworks by running `dotnet build` with no `--framework` specification.

11.3.3 Creating a NuGet package and checking the contents

When you build the NuGet package, it should contain both frameworks. To test this out, run `dotnet pack`. Browse to the folder that has the SampleEventSource.1.0.0.nupkg file, and change the extension to .zip. NuGet packages are essentially zip files organized in a particular way. Use your normal zip tool to see the contents.

The contents of SampleEventSource.1.0.0.nupkg should look like this:

- _rels
- .rels
- lib
- net46
  - SampleEventSource.exe
  - SampleEventSource.runtimeconfig.json
- netcoreapp2.0
  - SampleEventSource.dll
  - SampleEventSource.runtimeconfig.json
- [Content_Types].xml
- SampleEventSource.nuspec

In the NuGet package, the .nuspec file defines the contents, dependencies, metadata, and so on. The two frameworks supported by the application get their own folder and copy of the binary. In the case of the .NET Framework, the binary is in .exe form because this is an executable application. The .NET Core version of the binary is a .dll because it's not a self-contained application (see chapter 3).

11.3.4 Per-framework build options

One thing we've overlooked in the previous scenario is the MyEventSource.cs file. By default, all the .cs files in the project folder are included in the build. This means that MyEventSource.cs is being built even when you target the `net46` framework.

The build doesn't fail because .NET 4.6 has all of the `EventSource` features used by your code, but suppose the requirement for this application is that it has to work

on an older version of the .NET Framework, like 4.5. Change the framework moniker to `net45` as follows.

Listing 11.8 csproj with `net45` instead of `net46`

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFrameworks
      >netcoreapp2.0;net45</TargetFrameworks>
    <RuntimeFrameworkVersion
      Condition=" '$(TargetFramework)' == 'netcoreapp2.0' "
      >2.0.0-*</RuntimeFrameworkVersion>
  </PropertyGroup>

  <ItemGroup
    Condition=" '$(TargetFramework)' == 'net45' ">
    <Reference Include="System" />
    <Reference Include="Microsoft.CSharp" />
  </ItemGroup>

</Project>
```

Set to net45

Set to net45

Also, be sure to fix the preprocessor directives in the Program.cs file, as follows.

Listing 11.9 Program.cs using NET45 instead of NET46

```
using System;

#if NET45
using System.Diagnostics.Eventing;
#endif

namespace SampleEventSource
{
  public class Program
  {
#if NET45
    private static readonly Guid Provider =
      Guid.Parse("B695E411-F53B-4C72-9F81-2926B2EA233A");
#endif

    public static void Main(string[] args)
    {
#if NET45
      var eventProvider = new EventProvider(Provider);
      eventProvider.WriteMessageEvent("Program started");
#else
      MyEventSource.Instance.ProgramStart();
#endif

      // Do some work

#if NET45
      eventProvider.WriteMessageEvent("Program completed");
      eventProvider.Dispose();
```

```
#else
      MyEventSource.Instance.ProgramStop();
#endif
    }
  }
}
```

Try to build it, and you'll see the following errors.

Listing 11.10 Errors when building for .NET Framework 4.5

```
C:\dev\SampleEventSource\MyEventSource.cs(14,7): error CS0246: The type or
 namespace name 'Channel' could not be found (are you missing a using
 directive or an assembly reference?)
C:\dev\SampleEventSource\MyEventSource.cs(14,17): error CS0103: The name
 'EventChannel' does not exist in the current context
C:\dev\SampleEventSource\MyEventSource.cs(25,7): error CS0246: The type or
 namespace name 'Channel' could not be found (are you missing a using
 directive or an assembly reference?)
C:\dev\SampleEventSource\MyEventSource.cs(25,17): error CS0103: The name
 'EventChannel' does not exist in the current context

Compilation failed.
    0 Warning(s)
    4 Error(s)
```

You need to remove the MyEventSource.cs file from compilation when building for the `net45` framework. Change the csproj to exclude the MyEventSource.cs file from compilation under `net45`. You learned how to do this in chapter 3. The following listing shows how this would be done in your project.

Listing 11.11 csproj with framework-specific `buildOptions`

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFrameworks>netcoreapp2.0;net45</TargetFrameworks>
    <RuntimeFrameworkVersion
      Condition=" '$(TargetFramework)' == 'netcoreapp2.0' "
      >2.0.0-*</RuntimeFrameworkVersion>
  </PropertyGroup>

  <ItemGroup Condition=" '$(TargetFramework)' == 'net45' ">
    <Reference Include="System" />
    <Reference Include="Microsoft.CSharp" />
    <Compile Remove="MyEventSource.cs" />
  </ItemGroup>

</Project>
```

Add this line.

You should now be able to successfully build and run this application in either framework.

11.4 Runtime-specific code

In section 11.1 we looked at examples of .NET Core libraries taking a dependency on a native library, like libuv or zlib, to do some low-level operations with the operating system. You may need to do this in your library or application.

To do so, you'll need to define the runtimes you support in the `RuntimeIdentifiers` in the csproj, as follows.

Listing 11.12 Enumerating multiple runtimes in csproj

```
<PropertyGroup>
  <TargetFrameworks>netcoreapp2.0;net46</TargetFrameworks>
  <OutputType>Exe</OutputType>
  <RuntimeIdentifiers>osx.10.11-x64;ubuntu-x64</RuntimeIdentifiers>
</PropertyGroup>
```

To illustrate code that's OS-dependent, you'll attempt to get the process ID of the process your code is running in, without the help of .NET Core. If you peek into how .NET Core does it, you'll find the code that I'm using in this section (see https://github.com/dotnet/corefx).

To get the process ID on Windows, you can use the code in the following listing.

Listing 11.13 Interop.WindowsPid.cs—code to get the process ID on Windows

```
using System.Runtime.InteropServices;

internal partial class Interop
{
  internal partial class WindowsPid
  {
    [DllImport("api-ms-win-core-processthreads-l1-1-0.dll")]
    internal extern static uint GetCurrentProcessId();
  }
}
```

Note that this code doesn't have an implementation. It uses `DllImport` to make an interop call to a native assembly. The native assembly has a method called `GetCurrentProcessId` that does the real work.

Similarly, the following listing shows the code .NET Core uses to get the process ID on Linux systems.

Listing 11.14 Interop.LinuxPid.cs—code to get the process ID on Linux

```
using System.Runtime.InteropServices;

internal static partial class Interop
{
  internal static partial class LinuxPid
  {
    [DllImport("System.Native",
```

```
        EntryPoint="SystemNative_GetPid")]
    internal static extern int GetPid();
  }
}
```

The question is how you can use the Linux code on Linux runtimes and the Windows code on Windows runtimes. Given our discussion in the previous section on supporting multiple frameworks, you might think the answer is to use preprocessor directives and a per-runtime setting in the project file. Unfortunately, there are no extra build settings you can provide for specific runtimes. NuGet packages don't distinguish the runtime in the same way that they do frameworks.

That leaves detecting the operating system up to the code. Try this out by using the previous process ID code. First, create a new folder called Xplat, and open a command prompt in that folder. Run `dotnet new console`. Then create the Interop.WindowsPid.cs and Interop.LinuxPid.cs files, as listed earlier.

Now create a file called PidUtility.cs with the following code.

Listing 11.15 Contents of PidUtility.cs

```
using System;
using System.Runtime.InteropServices;

namespace Xplat
{
  public static class PidUtility
  {
    public static int GetProcessId()
    {
      var isWindows = RuntimeInformation.IsOSPlatform(OSPlatform.Windows);
      var isLinux = RuntimeInformation.IsOSPlatform(OSPlatform.Linux);

      if (isWindows)
        return (int)Interop.WindowsPid.GetCurrentProcessId();
      else if (isLinux)
        return Interop.LinuxPid.GetPid();
      else
        throw new PlatformNotSupportedException("Unsupported platform");
    }
  }
}
```

This utility class detects the OS at runtime and uses the appropriate implementation of the process ID interop class. To test it out, write a simple `Console.WriteLine` in the Program.cs file, as follows.

Listing 11.16 Contents of Program.cs

```
using System;

namespace Xplat
{
```

```
  public class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine($"My PID is {PidUtility.GetProcessId()}");
    }
  }
}
```

Do a `dotnet run`. If you're running on a Windows or Linux machine or a Docker container, you should see the process ID.

If you're writing a library, you should indicate in the csproj that you only support the two runtimes. This lets any projects that depend on yours know what runtimes they will function on. The following listing shows how to do this.

Listing 11.17 Xplat.csproj modified to indicate support for only two runtimes

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <RuntimeIdentifiers>win;linux</RuntimeIdentifiers>
  </PropertyGroup>

</Project>
```

Note that the `win` and `linux` runtimes are pretty broad categories. I picked them for demonstration purposes, but it may be necessary to be more specific about which operating systems the native code will work on.

Additional resources

To learn more about what we covered in this chapter, see the following resources:

- .NET Core GitHub repo—https://github.com/dotnet/corefx
- .NET Portability Analyzer—http://mng.bz/P5qN

Summary

In this chapter we looked at how to build applications that work differently depending on the framework or runtime in which they're used. We covered these key concepts:

- Using the .NET Portability Analyzer to assist in porting code between frameworks
- Using precompiler directives to build different code for different frameworks
- Creating code that uses OS-specific features

These are some important techniques to remember from this chapter:

- Precompiler directives can be used to optionally build code based on build properties.

- The .NET SDK `pack` command will generate NuGet packages that have all the frameworks you target.

Many of the early .NET Core projects undertaken by .NET Framework developers will involve porting existing code to .NET Core or .NET Standard. The .NET Portability Analyzer provides useful suggestions for these kinds of migrations. With the multiple framework support in .NET SDK, you can use newer features in .NET Core while still preserving functionality from existing applications.

You also learned about the flexibility in the .NET Core SDK for supporting multiple operating systems. This is useful when writing code that works with OS-specific libraries or features.

These two features in .NET Core—support for multiple frameworks and runtimes—are useful when porting existing projects. Whether you're moving from .NET Framework to .NET Core, Windows to Linux, or both, these features should give you the ability to tackle some of the more difficult issues encountered when converting a project to a new development platform.

.NET DEVELOPMENT

.NET Core IN ACTION

Dustin Metzgar

.NET Core is an open source framework that lets you write and run .NET applications on Linux and Mac, without giving up on Windows. Built for everything from lightweight web apps to industrial-strength distributed systems, it's perfect for deploying .NET servers to any cloud platform, including AWS and GCP.

.NET Core in Action introduces you to cross-platform development with .NET Core. This hands-on guide concentrates on new Core features as you walk through familiar tasks like testing, logging, data access, and networking. As you go, you'll explore modern architectures like microservices and cloud data storage, along with practical matters like performance profiling, localization, and signing assemblies.

What's Inside

- Choosing the right tools
- Testing, profiling, and debugging
- Interacting with web services
- Converting existing projects to .NET Core
- Creating and using NuGet packages

All examples are in C#.

Dustin Metzgar is a seasoned developer and architect involved in numerous .NET Core projects. Dustin works for Microsoft.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/dotnet-core-in-action

"A great on-ramp to the world of .NET and .NET Core. You'll learn the why, what, and how of building systems on this new platform."

—From the Foreword by Scott Hanselman, Microsoft

"Covers valuable use cases such as data access, web app development, and deployment to multiple platforms."

—Viorel Moisei
Gabriels Technology Solutions

"Teaches you to write code that ports across all platforms; also includes tips for porting legacy code to .NET Core."

—Eddy Vluggen, Cadac

"Covers all the new tools and features of .NET Core. Brain-friendly."

—Tiklu Ganguly, ITC Infotech

ISBN-13: 978-1-61729-427-3
ISBN-10: 1-61729-427-6
54499
9 781617 294273

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]