.NET Core IN ACTION

Dustin Metzgar
Foreword by Scott Hanselman

SAMPLE CHAPTER

MANNING

.NET Core in Action

by Dustin Metzgar

Sample Chapter 11

brief contents

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.

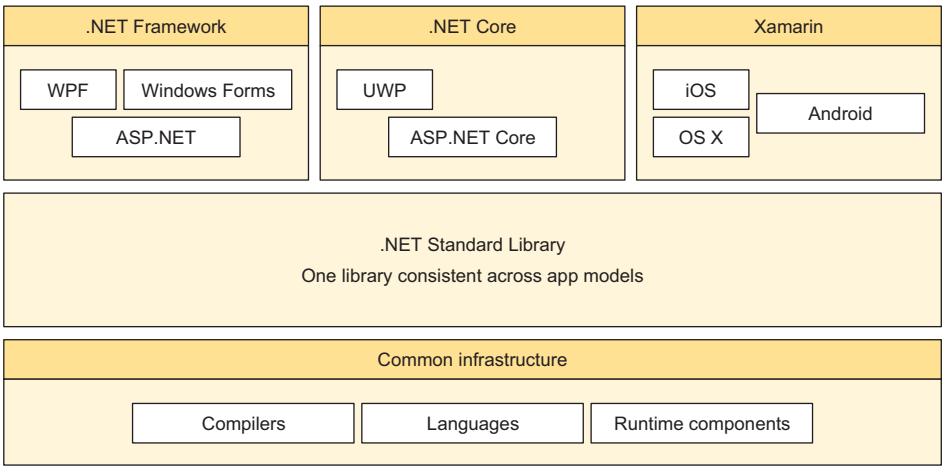

Figure 11.1 .NET Framework, .NET Core, and Xamarin are all different frameworks that support the .NET Standard Library.

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.

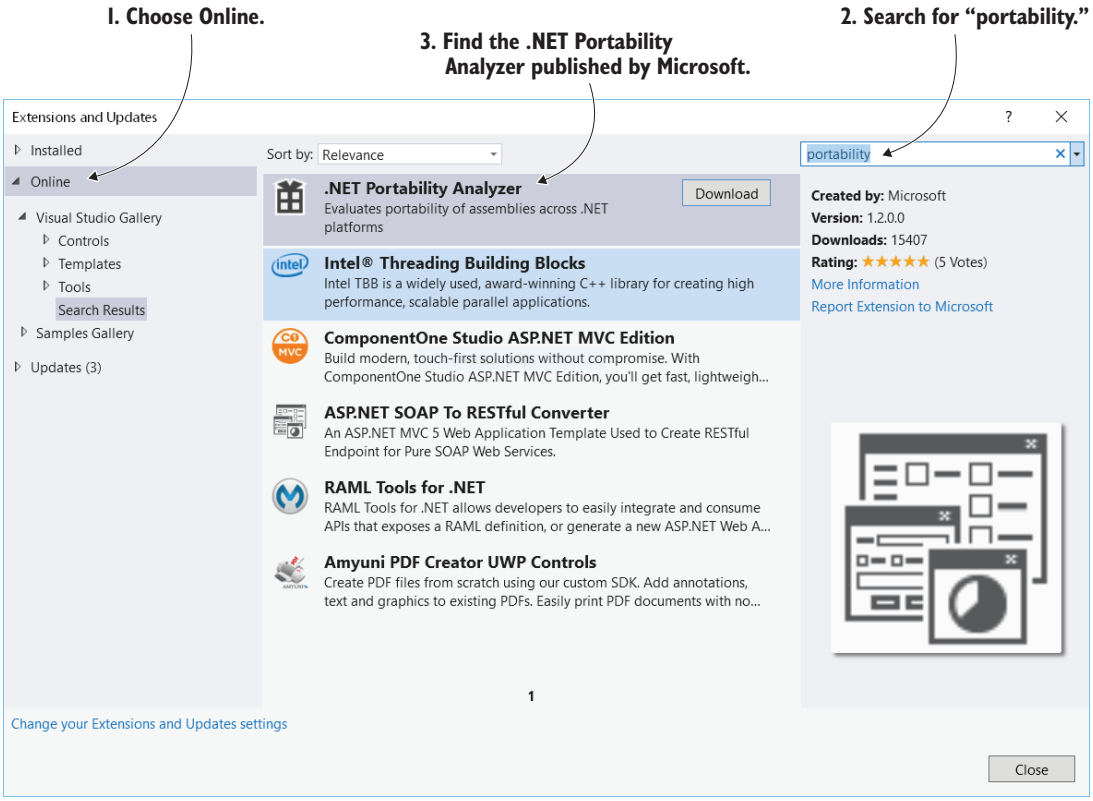

Figure 11.2 Search for the .NET Portability Analyzer.

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.

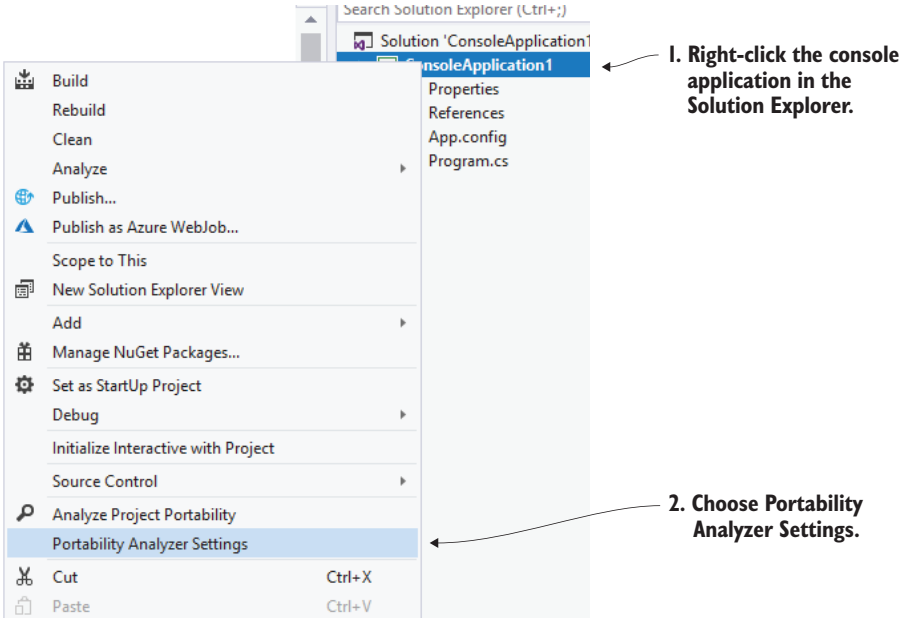

Figure 11.3 Open the settings for the Portability Analyzer.

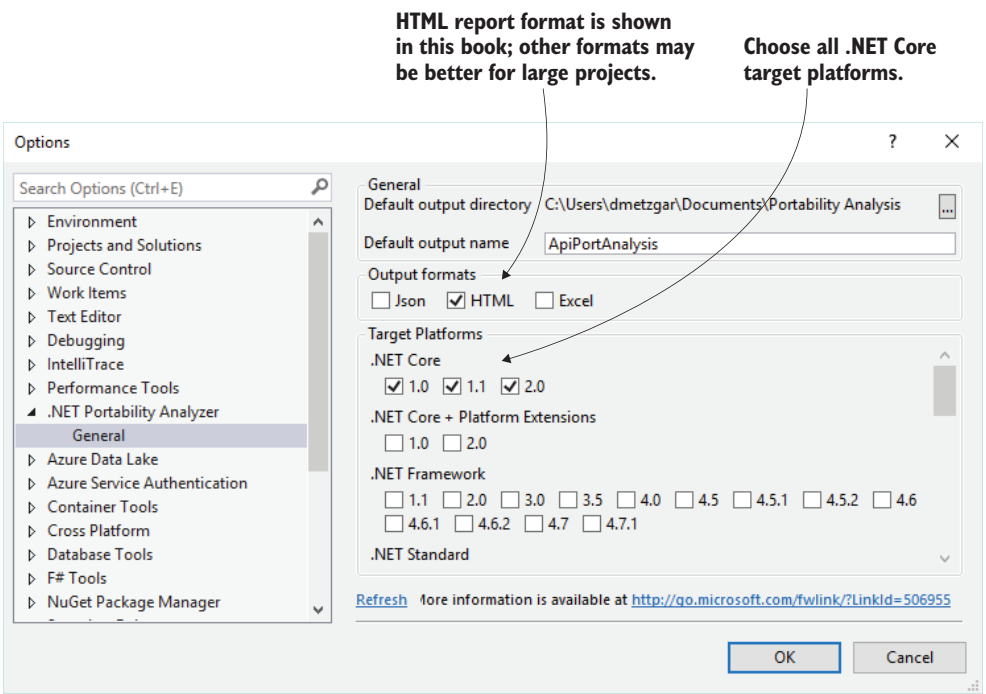

Figure 11.4 Choose all .NET Core Target Platforms in the Portability Analyzer settings.

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

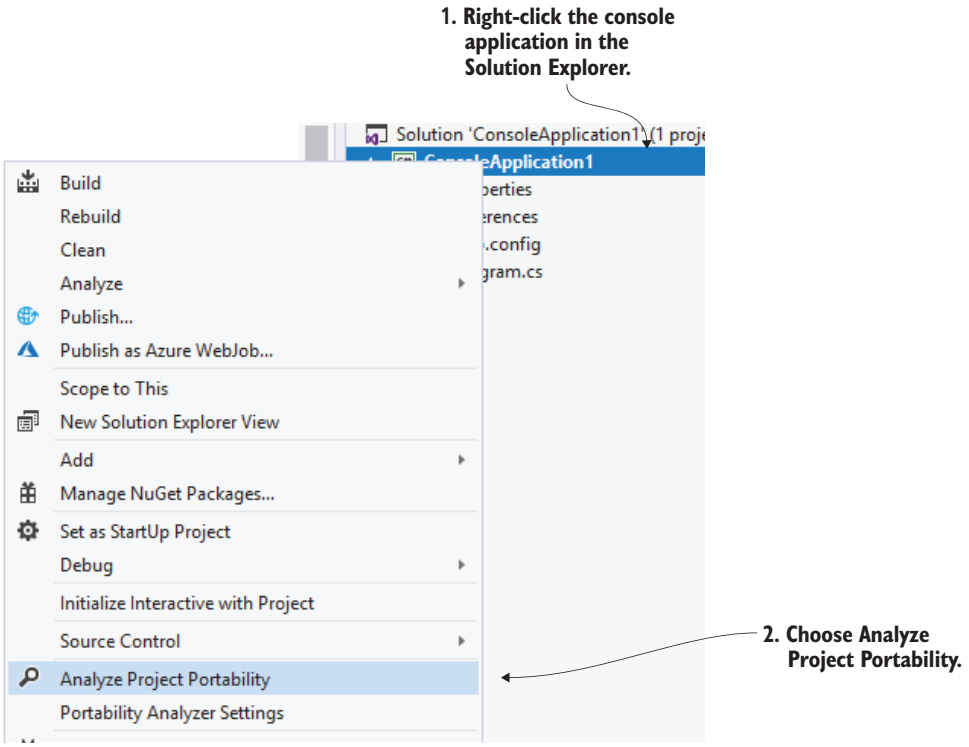

Figure 11.5 Run the Portability Analyzer from the right-click menu.

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

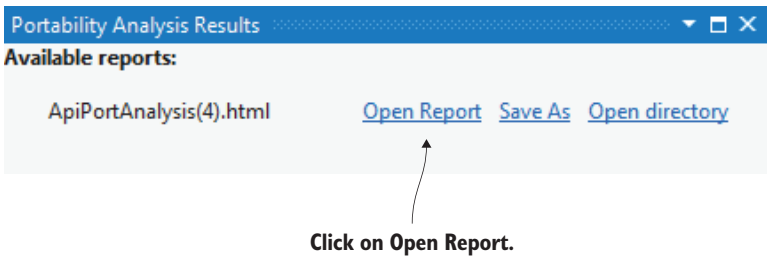

Figure 11.6 Portability Analyzer Results pane

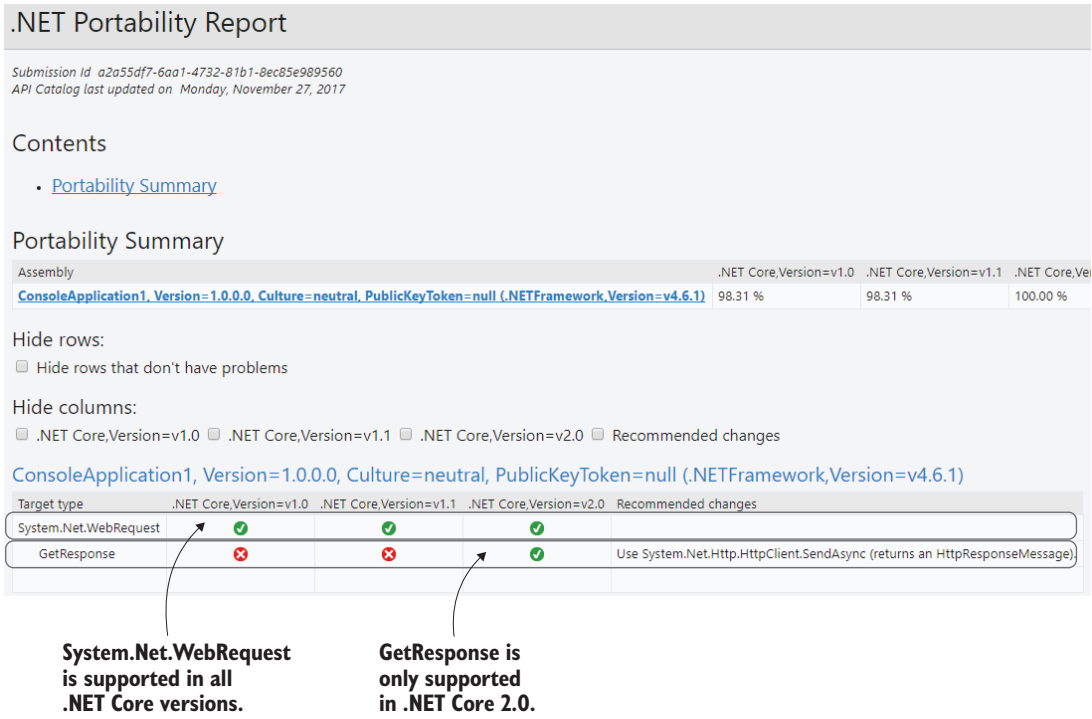

Figure 11.7 Portability analysis of the sample code

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.

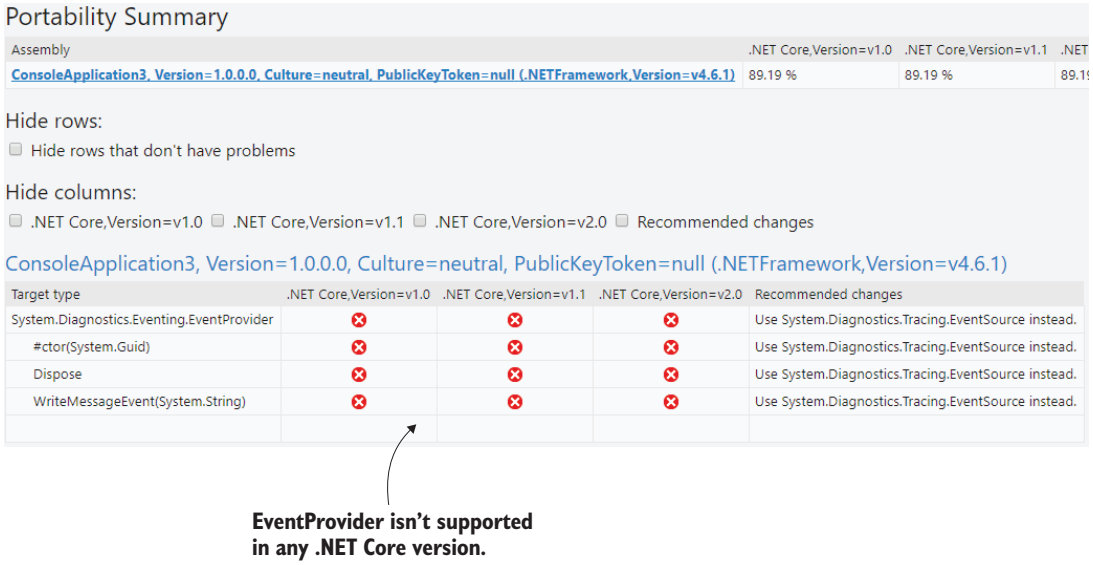

Figure 11.8 Portability analysis of the sample code using `EventProvider`

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]