

7

Packaging and Distributing .NET Types

This chapter is about how C# keywords are related to .NET types, and about the relationship between namespaces and assemblies. You'll also become familiar with how to package and publish your .NET apps and libraries for cross-platform use, how to use legacy .NET Framework libraries in .NET libraries, and the possibility of porting legacy .NET Framework code bases to modern .NET.

This chapter covers the following topics:

- The road to .NET 7
- Understanding .NET components
- Publishing your applications for deployment
- Decompiling .NET assemblies
- Packaging your libraries for NuGet distribution
- Porting from .NET Framework to modern .NET
- Working with preview features

The road to .NET 7

This part of the book is about the functionality in the **Base Class Library (BCL)** APIs provided by .NET and how to reuse functionality across all the different .NET platforms using .NET Standard.

First, we will review the route to this point and why it is important to understand the past.

.NET Core 2.0 and later's support for a minimum of .NET Standard 2.0 is important because it provides many of the APIs that were missing from the first version of .NET Core. The 15 years' worth of libraries and applications that .NET Framework developers had available to them that are relevant for modern development have now been migrated to .NET and can run cross-platform on macOS and Linux variants, as well as on Windows.

.NET Standard 2.1 added about 3,000 new APIs. Some of those APIs need runtime changes that would break backward compatibility, so .NET Framework 4.8 only implements .NET Standard 2.0. .NET Core 3.0, Xamarin, Mono, and Unity implement .NET Standard 2.1.

.NET 5 removed the need for .NET Standard if all your projects could use .NET 5. The same applies to .NET 6 and .NET 7. Since you might still need to create class libraries for legacy .NET Framework projects or legacy Xamarin mobile apps, there is still a need to create .NET Standard 2.0 and 2.1 class libraries. In March 2021, I surveyed professional developers, and half still needed to create .NET Standard 2.0 compliant class libraries.

Now that .NET 6 and .NET 7 have full support for mobile and desktop apps built using .NET MAUI, the need for .NET Standard has been further reduced.

To summarize the progress that .NET has made over the past five years, I have compared the major .NET Core and modern .NET versions with the equivalent .NET Framework versions in the following list:

- **.NET Core 1.x:** Much smaller API compared to .NET Framework 4.6.1, which was the current version in March 2016.
- **.NET Core 2.x:** Reached API parity with .NET Framework 4.7.1 for modern APIs because they both implement .NET Standard 2.0.
- **.NET Core 3.x:** Larger API compared to .NET Framework for modern APIs because .NET Framework 4.8 does not implement .NET Standard 2.1.
- **.NET 5:** Even larger API compared to .NET Framework 4.8 for modern APIs, with much-improved performance.
- **.NET 6:** Continued improvements to performance and expanded APIs. Optional support for mobile apps in .NET MAUI added in May 2022.
- **.NET 7:** Final unification with the support for mobile apps in .NET MAUI.

.NET Core 1.0

.NET Core 1.0 was released in June 2016 and focused on implementing an API suitable for building modern cross-platform apps, including web and cloud applications and services for Linux using ASP.NET Core.

.NET Core 1.1

.NET Core 1.1 was released in November 2016 and focused on fixing bugs, increasing the number of Linux distributions supported, supporting .NET Standard 1.6, and improving performance, especially with ASP.NET Core for web apps and services.

.NET Core 2.0

.NET Core 2.0 was released in August 2017 and focused on implementing .NET Standard 2.0, the ability to reference .NET Framework libraries, and more performance improvements.

The third edition of this book was published in November 2017, so it covered up to .NET Core 2.0 and .NET Core for **Universal Windows Platform (UWP)** apps.

.NET Core 2.1

.NET Core 2.1 was released in May 2018 and focused on an extendable tooling system, adding new types like `Span<T>`, new APIs for cryptography and compression, a Windows Compatibility Pack with an additional 20,000 APIs to help port old Windows applications, Entity Framework Core value conversions, LINQ GroupBy conversions, data seeding, query types, and even more performance improvements, including the topics listed in the following table:

Feature	Chapter	Topic
Spans	8	Working with spans, indexes, and ranges
Brotli compression	9	Compressing with the Brotli algorithm
EF Core Lazy loading	10	Enabling lazy loading
EF Core Data seeding	10	Understanding data seeding

.NET Core 2.2

.NET Core 2.2 was released in December 2018 and focused on diagnostic improvements for the runtime, optional tiered compilation, and adding new features to ASP.NET Core and Entity Framework Core like spatial data support using types from the `NetTopologySuite` (NTS) library, query tags, and collections of owned entities.

.NET Core 3.0

.NET Core 3.0 was released in September 2019 and focused on adding support for building Windows desktop applications using Windows Forms (2001), **Windows Presentation Foundation** (WPF; 2006), and Entity Framework 6.3, side-by-side and app-local deployments, a fast JSON reader, serial port access and other pinout access for **Internet of Things** (IoT) solutions, and tiered compilation by default, including the topics listed in the following table:

Feature	Chapter	Topic
Embedding .NET in-app	7	Publishing your applications for deployment
Index and Range	8	Working with spans, indexes, and ranges
<code>System.Text.Json</code>	9	High-performance JSON processing

The fourth edition of this book was published in October 2019, so it covered some of the new APIs added in later versions up to .NET Core 3.0.

.NET Core 3.1

.NET Core 3.1 was released in December 2019 and focused on bug fixes and refinements so that it could be a **Long Term Support** (LTS) release, not losing support until December 2022.

.NET 5.0

.NET 5.0 was released in November 2020 and focused on unifying the various .NET platforms except mobile, refining the platform, and improving performance, including the topics listed in the following table:

Feature	Chapter	Topic
Half type	8	Working with numbers
Regular expression performance improvements	8	Regular expression performance improvements
System.Text.Json improvements	9	High-performance JSON processing
EF Core generated SQL	10	Getting the generated SQL
EF Core Filtered Include	10	Filtering included entities
EF Core Scaffold-DbContext now singularizes using Humanizer	10	Scaffolding models using an existing database

.NET 6.0

.NET 6.0 was released in November 2021 and focused on adding more features to EF Core for data management, new types for working with dates and times, and improving performance, including the topics listed in the following table:

Feature	Chapter	Topic
Check .NET SDK status	7	Checking your .NET SDKs for updates
Support for Apple Silicon	7	Creating a console application to publish
Link trim mode as default	7	Reducing the size of apps using app trimming
EnsureCapacity for List<T>	8	Improving performance by ensuring the capacity of a collection
Low-level file API using RandomAccess	9	Reading and writing with random access handles
EF Core configure conventions	10	Configuring preconvention models
New LINQ methods	11	Building LINQ expressions with the Enumerable class
TryGetNonEnumeratedCount	11	Aggregating sequences

.NET 7.0

.NET 7.0 was released in November 2022 and focused on unifying with the mobile platform, adding more features like string syntax coloring and IntelliSense, support for creating and extracting tar archives, and improving performance of inserts and updates with EF Core, including the topics listed in the following table:

Feature	Chapter	Topic
[StringSyntax] attribute	8	Activating regular expression syntax coloring
[GeneratedRegex] attribute	8	Improving regular expression performance with source generators
Tar archive support	9	Working with tar archives
ExecuteUpdate and ExecuteDelete	10	More efficient updates and deletes
Order and OrderDescending	11	Sorting by the item itself

Improving performance with .NET 5 and later

Microsoft has made significant improvements to performance in the past few years. You can read detailed blog posts at the following links:

<https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5/>

<https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-6/>

https://devblogs.microsoft.com/dotnet/performance_improvements_in_net_7/

Checking your .NET SDKs for updates

With .NET 6, Microsoft added a command to check the versions of .NET SDKs and runtimes that you have installed and warn you if any need updating. For example, enter the following command:

```
dotnet sdk check
```

You will see results, including the status of available updates, as shown in the following partial output:

```
.NET SDKs:
Version          Status
-----
3.1.421          .NET Core 3.1 is going out of support soon.
5.0.406          .NET 5.0 is out of support.
6.0.300          Patch 6.0.301 is available.
7.0.100          Up to date.
```

Understanding .NET components

.NET is made up of several pieces, which are shown in the following list:

- **Language compilers:** These turn your source code written with languages such as C#, F#, and Visual Basic into **intermediate language (IL)** code stored in assemblies. With C# 6.0 and later, Microsoft switched to an open-source rewritten compiler known as Roslyn that is also used by Visual Basic.
- **Common Language Runtime (CoreCLR):** This runtime loads assemblies, compiles the IL code stored in them into native code instructions for your computer's CPU, and executes the code within an environment that manages resources such as threads and memory.
- **Base Class Libraries (BCL or CoreFX):** These are prebuilt assemblies of types packaged and distributed using NuGet for performing common tasks when building applications. You can use them to quickly build anything you want, rather like combining LEGO™ pieces.

Assemblies, NuGet packages, and namespaces

An **assembly** is where a type is stored in the filesystem. Assemblies are a mechanism for deploying code. For example, the `System.Data.dll` assembly contains types for managing data. To use types in other assemblies, they must be referenced. Assemblies can be static (pre-created) or dynamic (generated at runtime). Dynamic assemblies are an advanced feature that we will not cover in this book. Assemblies can be compiled into a single file as a DLL (class library) or an EXE (console app).

Assemblies are distributed as **NuGet packages**, which are files downloadable from public online feeds and can contain multiple assemblies and other resources. You will also hear about **project SDKs**, **workloads**, and **platforms**, which are combinations of NuGet packages.

Microsoft's NuGet feed is found here: <https://www.nuget.org/>.

What is a namespace?

A namespace is the address of a type. Namespaces are a mechanism to uniquely identify a type by requiring a full address rather than just a short name. In the real world, *Bob of 34 Sycamore Street* is different from *Bob of 12 Willow Drive*.

In .NET, the `IActionFilter` interface of the `System.Web.Mvc` namespace is different from the `IActionFilter` interface of the `System.Web.Http.Filters` namespace.

Dependent assemblies

If an assembly is compiled as a class library and provides types for other assemblies to use, then it has the file extension `.dll` (dynamic link library), and it cannot be executed standalone.

Likewise, if an assembly is compiled as an application, then it has the file extension `.exe` (executable) and can be executed standalone. Before .NET Core 3.0, console apps were compiled to `.dll` files and had to be executed by the `dotnet run` command or a host executable.

Any assembly can reference one or more class library assemblies as dependencies, but you cannot have circular references. So, assembly *B* cannot reference assembly *A* if assembly *A* already references assembly *B*. The compiler will warn you if you attempt to add a dependency reference that would cause a circular reference. Circular references are often a warning sign of poor code design. If you are sure that you need a circular reference, then use an interface to solve it.

Microsoft .NET project SDKs

By default, console applications have a dependency reference on the Microsoft .NET project SDK. This platform contains thousands of types in NuGet packages that almost all applications would need, such as the `System.Int32` and `System.String` types.

When using .NET, you reference the dependency assemblies, NuGet packages, and platforms that your application needs in a project file.

Let's explore the relationship between assemblies and namespaces:

1. Use your preferred code editor to create a new project, as defined in the following list:
 - Project template: **Console App/console**
 - Project file and folder: **AssembliesAndNamespaces**
 - Workspace/solution file and folder: **Chapter07**
2. Open `AssembliesAndNamespaces.csproj` and note that it is a typical project file for a .NET application, as shown in the following markup:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

3. After the `<PropertyGroup>` section, add a new `<ItemGroup>` section to statically import `System.Console` for all C# files using the implicit usings .NET SDK feature, as shown in the following markup:

```
<ItemGroup>
  <Using Include="System.Console" Static="true" />
</ItemGroup>
```

Namespaces and types in assemblies

Many common .NET types are in the `System.Runtime.dll` assembly. There is not always a one-to-one mapping between assemblies and namespaces. A single assembly can contain many namespaces and a namespace can be defined in many assemblies. You can see the relationship between some assemblies and the namespaces that they supply types for, as shown in the following table:

Assembly	Example namespaces	Example types
<code>System.Runtime.dll</code>	<code>System, System.Collections, System.Collections.Generic</code>	<code>Int32, String, IEnumerable<T></code>
<code>System.Console.dll</code>	<code>System</code>	<code>Console</code>
<code>System.Threading.dll</code>	<code>System.Threading</code>	<code>Interlocked, Monitor, Mutex</code>
<code>System.Xml.XDocument.dll</code>	<code>System.Xml.Linq</code>	<code>XDocument, XElement, XNode</code>

NuGet packages

.NET is split into a set of packages, distributed using a Microsoft-supported package management technology named NuGet. Each of these packages represents a single assembly of the same name. For example, the `System.Collections` package contains the `System.Collections.dll` assembly.

The following are the benefits of packages:

- Packages can be easily distributed on public feeds.
- Packages can be reused.
- Packages can ship on their own schedule.
- Packages can be tested independently of other packages.
- Packages can support different OSes and CPUs by including multiple versions of the same assembly built for different OSes and CPUs.
- Packages can have dependencies specific to only one library.
- Apps are smaller because unreferenced packages aren't part of the distribution. The following table lists some of the more important packages and their important types:

Package	Important types
<code>System.Runtime</code>	<code>Object, String, Int32, Array</code>
<code>System.Collections</code>	<code>List<T>, Dictionary< TKey, TValue ></code>
<code>System.Net.Http</code>	<code>HttpClient, HttpResponseMessage</code>
<code>System.IO.FileSystem</code>	<code>File, Directory</code>
<code>System.Reflection</code>	<code>Assembly, TypeInfo, MethodInfo</code>

Understanding frameworks

There is a two-way relationship between frameworks and packages. Packages define the APIs, while frameworks group packages. A framework without any packages would not define any APIs.

.NET packages each support a set of frameworks. For example, the `System.IO.FileSystem` package version 4.3.0 supports the following frameworks:

- .NET Standard, version 1.3 or later
- .NET Framework, version 4.6 or later
- Six Mono and Xamarin platforms (for example, Xamarin.iOS 1.0)



More Information: You can read the details at the following link: <https://www.nuget.org/packages/System.IO.FileSystem/>.

Importing a namespace to use a type

Let's explore how namespaces are related to assemblies and types:

1. In the `AssembliesAndNamespaces` project, in `Program.cs`, delete the existing statements and then enter the following code:

```
XDocument doc = new();
```

2. Build the project and note the compiler error message, as shown in the following output:

```
The type or namespace name 'XDocument' could not be found (are you
missing a using directive or an assembly reference?)
```

The `XDocument` type is not recognized because we have not told the compiler what the namespace of the type is. Although this project already has a reference to the assembly that contains the type, we also need to either prefix the type name with its namespace or import the namespace.

3. Click inside the `XDocument` class name. Your code editor displays a light bulb, showing that it recognizes the type and can automatically fix the problem for you.
4. Click the light bulb, and select `using System.Xml.Linq;` from the menu.

This will *import the namespace* by adding a `using` statement to the top of the file. Once a namespace is imported at the top of a code file, then all the types within the namespace are available for use in that code file by just typing their name without the type name needing to be fully qualified by prefixing it with its namespace.

Sometimes I like to add a comment with a type name after importing a namespace to remind me which types require me to import that namespace, as shown in the following code:

```
using System.Xml.Linq; // XDocument
```

Relating C# keywords to .NET types

One of the common questions I get from new C# programmers is, “What is the difference between `string` with a lowercase s and `String` with an uppercase S?”

The short answer is easy: none. The long answer is that all C# type keywords like `string` or `int` are aliases for a .NET type in a class library assembly.

When you use the `string` keyword, the compiler recognizes it as a `System.String` type. When you use the `int` type, the compiler recognizes it as a `System.Int32` type.

Let's see this in action with some code:

1. In `Program.cs`, declare two variables to hold `string` values, one using lowercase `string` and one using uppercase `String`, as shown in the following code:

```
string s1 = "Hello";
String s2 = "World";
WriteLine($"{s1} {s2}");
```

2. Run the code, and note that at the moment, they both work equally well, and literally mean the same thing.
3. In `AssembliesAndNamespaces.csproj`, add entries to prevent the `System` namespace from being globally imported, as shown in the following markup:

```
<ItemGroup>
  <Using Remove="System" />
</ItemGroup>
```

4. In `Program.cs`, note the compiler error message, as shown in the following output:

`The type or namespace name 'String' could not be found (are you missing a using directive or an assembly reference?)`
5. At the top of `Program.cs`, import the `System` namespace with a `using` statement that will fix the error, as shown in the following code:

```
using System; // String
```



Good Practice: When you have a choice, use the C# keyword instead of the actual type because the keywords do not need the namespace to be imported.

Mapping C# aliases to .NET types

The following table shows the 18 C# type keywords along with their actual .NET types:

Keyword	.NET type	Keyword	.NET type
string	System.String	char	System.Char
sbyte	System.SByte	byte	System.Byte
short	System.Int16	ushort	System.UInt16
int	System.Int32	uint	System.UInt32
long	System.Int64	ulong	System.UInt64
nint	System.IntPtr	nuint	System.UIntPtr
float	System.Single	double	System.Double
decimal	System.Decimal	bool	System.Boolean
object	System.Object	dynamic	System.Dynamic.DynamicObject

Other .NET programming language compilers can do the same thing. For example, the Visual Basic .NET language has a type named `Integer` that is its alias for `System.Int32`.

Understanding native-sized integers

C# 9 introduced the `nint` and `nuint` keyword aliases for **native-sized integers**, meaning that the storage size for the integer value is platform-specific. They store a 32-bit integer in a 32-bit process and `sizeof()` returns 4 bytes; they store a 64-bit integer in a 64-bit process and `sizeof()` returns 8 bytes. The aliases represent pointers to the integer value in memory, which is why their .NET names are `IntPtr` and `UIntPtr`. The actual storage type will be either `System.Int32` or `System.Int64` depending on the process.

In a 64-bit process, the following code:

```
WriteLine($"int.MaxValue = {int.MaxValue:N0}");
WriteLine($"nint.MaxValue = {nint.MaxValue:N0}");
```

produces this output:

```
int.MaxValue = 2,147,483,647
nint.MaxValue = 9,223,372,036,854,775,807
```

Revealing the location of a type

Code editors provide built-in documentation for .NET types. Let's explore:

1. Right-click inside `XDocument` and choose **Go to Definition**.

2. Navigate to the top of the code file and note the assembly filename is `System.Xml.XDocument.dll`, but the class is in the `System.Xml.Linq` namespace, as shown in *Figure 7.1*:

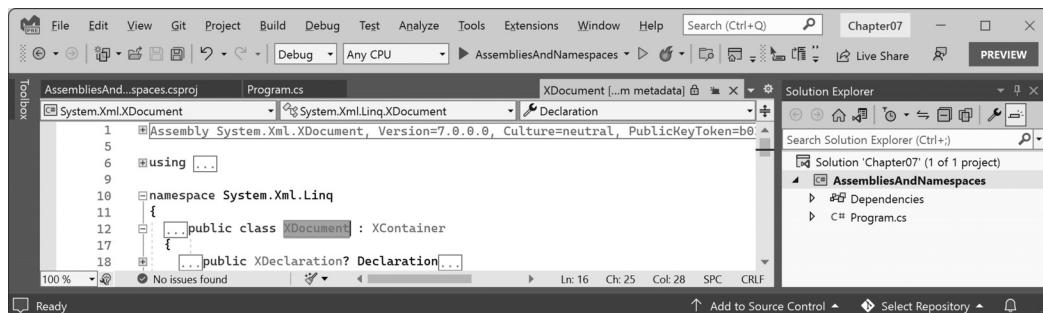


Figure 7.1: Assembly and namespace that contains the `XDocument` type

3. Close the `XDocument [from metadata]` tab.
4. Right-click inside `string` or `String` and choose **Go to Definition**.
5. Navigate to the top of the code file and note the assembly filename is `System.Runtime.dll` but the class is in the `System` namespace.

Your code editor is technically lying to you. If you remember when we wrote code in *Chapter 2, Speaking C#*, when we revealed the extent of the C# vocabulary, we discovered that the `System.Runtime.dll` assembly contains zero types.

What the `System.Runtime.dll` assembly does contain are type-forwarders. These are special types that appear to exist in an assembly but are implemented elsewhere. In this case, they are implemented deep inside the .NET runtime using highly optimized code.

Sharing code with legacy platforms using .NET Standard

Before .NET Standard, there were **Portable Class Libraries (PCLs)**. With PCLs, you could create a library of code and explicitly specify which platforms you want the library to support, such as Xamarin, Silverlight, and Windows 8. Your library could then use the intersection of APIs that are supported by the specified platforms.

Microsoft realized that this is unsustainable, so they created .NET Standard—a single API that all future .NET platforms would support. There are older versions of .NET Standard, but .NET Standard 2.0 was an attempt to unify all important recent .NET platforms. .NET Standard 2.1 was released in late 2019 but only .NET Core 3.0 and that year's version of Xamarin support its new features. For the rest of this book, I will use the term .NET Standard to mean .NET Standard 2.0.

.NET Standard is like HTML5 in that they are both standards that a platform should support. Just as Google's Chrome browser and Microsoft's Edge browser implement the HTML5 standard, .NET Core, .NET Framework, and Xamarin all implement .NET Standard. If you want to create a library of types that will work across variants of legacy .NET, you can do so most easily with .NET Standard.



Good Practice: Since many of the API additions in .NET Standard 2.1 required runtime changes, and .NET Framework is Microsoft's legacy platform that needs to remain as unchanging as possible, .NET Framework 4.8 remained on .NET Standard 2.0 rather than implementing .NET Standard 2.1. If you need to support .NET Framework customers, then you should create class libraries on .NET Standard 2.0 even though it is not the latest and does not support all the recent language and BCL new features.

Your choice of which .NET Standard version to target comes down to a balance between maximizing platform support and available functionality. A lower version supports more platforms but has a smaller set of APIs. A higher version supports fewer platforms but has a larger set of APIs. Generally, you should choose the lowest version that supports all the APIs that you need.

Understanding defaults for class libraries with different SDKs

When using the dotnet SDK tool to create a class library, it might be useful to know which target framework will be used by default, as shown in the following table:

SDK	Default target framework for new class libraries
.NET Core 3.1	netstandard2.0
.NET 6	net6.0
.NET 7	net7.0

Of course, just because a class library targets a specific version of .NET by default does not mean you cannot change it after creating a class library project using the default template.

You can manually set the target framework to a value that supports the projects that need to reference that library, as shown in the following table:

Class library target framework	Can be used by projects that target
netstandard2.0	.NET Framework 4.6.1 or later, .NET Core 2.0 or later, .NET 5.0 or later, Mono 5.4 or later, Xamarin.Android 8.0 or later, Xamarin.iOS 10.14 or later
netstandard2.1	.NET Core 3.0 or later, .NET 5.0 or later, Mono 6.4 or later, Xamarin.Android 10.0 or later, Xamarin.iOS 12.16 or later
net6.0	.NET 6.0 or later
net7.0	.NET 7.0 or later



Good Practice: Always check the target framework of a class library and then manually change it to something more appropriate if necessary. Make a conscious decision about what it should be rather than accepting the default.

Creating a .NET Standard 2.0 class library

We will create a class library using .NET Standard 2.0 so that it can be used across all important .NET legacy platforms and cross-platform on Windows, macOS, and Linux operating systems, while also having access to a wide set of .NET APIs:

1. Use your preferred code editor to add a new **Class Library**/classlib project named **SharedLibrary** that targets .NET Standard 2.0 to the **Chapter07** solution/workspace:

- If you use Visual Studio 2022, when prompted for the **Target Framework**, select **.NET Standard 2.0**, and then set the startup project for the solution to the current selection.
- If you use Visual Studio Code, include a switch to target .NET Standard 2.0, as shown in the following command, and then select **SharedLibrary** as the active OmniSharp project:

```
dotnet new classlib -f netstandard2.0
```



Good Practice: If you need to create types that use new features in .NET 7.0, as well as types that only use .NET Standard 2.0 features, then you can create two separate class libraries: one targeting .NET Standard 2.0 and one targeting .NET 7.0.

An alternative to manually creating two class libraries is to create one that supports **multi-targeting**. If you would like me to add a section about multi-targeting to the next edition, please let me know. You can read about multi-targeting here: <https://docs.microsoft.com/en-us/dotnet/standard/library-guidance/cross-platform-targeting#multi-targeting>.

Controlling the .NET SDK

By default, executing `dotnet` commands uses the most recent installed .NET SDK. There may be times when you want to control which SDK is used.

For example, one reader of the fourth edition wanted their experience to match the book steps that use the .NET Core 3.1 SDK. But they had installed the .NET 5.0 SDK as well and that was being used by default. As described in the previous section, the behavior when creating new class libraries changed to target .NET 5.0 instead of .NET Standard 2.0, and that confused the reader.

You can control the .NET SDK used by default by using a `global.json` file. The `dotnet` command searches the current folder and ancestor folders for a `global.json` file.

You do not need to complete the following steps, but if you want to try and do not already have .NET 6.0 SDK installed then you can install it from the following link:

<https://dotnet.microsoft.com/download/dotnet/6.0>

1. Create a subdirectory/folder in the **Chapter07** folder named **ControlSDK**.
2. On Windows, start **Command Prompt** or **Windows Terminal**. On macOS, start **Terminal**. If you are using Visual Studio Code, then you can use the integrated terminal.

3. In the ControlSDK folder, at the command prompt or terminal, enter a command to list the installed .NET SDKs, as shown in the following command:

```
dotnet --list-sdks
```

4. Note the results and the version number of the latest .NET 6 SDK installed, as shown in the following output:

```
3.1.416 [C:\Program Files\dotnet\sdk]
6.0.200 [C:\Program Files\dotnet\sdk]
7.0.100 [C:\Program Files\dotnet\sdk]
```

5. Create a `global.json` file that forces the use of the latest .NET Core 6.0 SDK that you have installed (which might be later than mine), as shown in the following command:

```
dotnet new globaljson --sdk-version 6.0.200
```

6. Open the `global.json` file and review its contents, as shown in the following markup:

```
{
  "sdk": {
    "version": "6.0.200"
  }
}
```

7. In the ControlSDK folder, at the command prompt or terminal, enter a command to create a class library project, as shown in the following command:

```
dotnet new classlib
```

8. If you do not have the .NET 6.0 SDK installed then you will see an error, as shown in the following output:

```
Could not execute because the application was not found or a compatible
.NET SDK is not installed.
```

9. If you do have the .NET 6.0 SDK installed, then a class library project will be created that targets .NET 6.0 by default.

Publishing your code for deployment

If you write a novel and you want other people to read it, you must publish it.

Most developers write code for other developers to use in their own projects, or for users to run as an app. To do so, you must publish your code as packaged class libraries or executable applications.

There are three ways to publish and deploy a .NET application. They are:

- **Framework-dependent deployment (FDD)**
- **Framework-dependent executable (FDE)**
- **Self-contained**

If you choose to deploy your application and its package dependencies, but not .NET itself, then you rely on .NET already being on the target computer. This works well for web applications deployed to a server because .NET and lots of other web applications are likely already on the server.

Framework-dependent deployment (FDD) means you deploy a DLL that must be executed by the dotnet command-line tool. **Framework-dependent executables (FDE)** means you deploy an EXE that can be run directly from the command line. Both require the appropriate version of the .NET runtime to be already installed on the system.

Sometimes, you want to be able to give someone a USB stick containing your application and know that it can execute on their computer. You want to perform a self-contained deployment. While the size of the deployment files will be larger, you'll know that it will work.

Creating a console app to publish

Let's explore how to publish a console app:

1. Use your preferred code editor to add a new **Console App/console** project named **DotNetEverywhere** to the **Chapter07** solution/workspace:
 - In Visual Studio Code, select **DotNetEverywhere** as the active OmniSharp project. When you see the pop-up warning message saying that required assets are missing, click **Yes** to add them.
2. Modify the project file to statically import the **System.Console** class in all C# files.
3. In **Program.cs**, delete the existing statements and then add a statement to output a message saying the console app can run everywhere and some information about the operating system, as shown in the following code:

```
WriteLine("I can run everywhere!");
WriteLine($"OS Version is {Environment.OSVersion}.");

if (OperatingSystem.IsMacOS())
{
    WriteLine("I am macOS.");
}
else if (OperatingSystem.IsWindowsVersionAtLeast(major: 10, build: 22000))
{
    WriteLine("I am Windows 11.");
}
else if (OperatingSystem.IsWindowsVersionAtLeast(major: 10))
{
    WriteLine("I am Windows 10.");
}
else
{
    WriteLine("I am some other mysterious OS.");
}
```

```
    }
    WriteLine("Press ENTER to stop me.");
    ReadLine();
```

4. Run the console app and note the results when run on Windows 11, as shown in the following output:

```
I can run everywhere!
OS Version is Microsoft Windows NT 10.0.22000.0.
I am Windows 11.
Press ENTER to stop me.
```

5. Open DotNetEverywhere.csproj and add the runtime identifiers to target three operating systems inside the `<PropertyGroup>` element, as shown highlighted in the following markup:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <RuntimeIdentifiers>
      win10-x64;osx-x64;osx.11.0-arm64;linux-x64;linux-arm64
    </RuntimeIdentifiers>
  </PropertyGroup>

</Project>
```

The highlighted items are as follows:

- The `win10-x64` RID value means Windows 10 or Windows Server 2016 64-bit. You could also use the `win10-arm64` RID value to deploy to a Microsoft Surface Pro X, Surface Pro 9 (SQ 3), or Windows Dev Kit 2023.
- The `osx-x64` RID value means macOS Sierra 10.12 or later. You can also specify version-specific RID values like `osx.10.15-x64` (Catalina), `osx.13.0-x64` (Ventura on Intel), or `osx.13.0-arm64` (Ventura on Apple Silicon).
- The `linux-x64` RID value means most desktop distributions of Linux, like Ubuntu, CentOS, Debian, or Fedora. Use `linux-arm` for Raspbian or Raspberry Pi OS 32-bit. Use `linux-arm64` for a Raspberry Pi running Ubuntu 64-bit.



There are two elements that you can use to specify runtime identifiers. Use `<RuntimeIdentifier>` if you only need to specify one. Use `<RuntimeIdentifiers>` if you need to specify multiple, as we did in the preceding example. If you use the wrong one, then the compiler will give an error and it can be difficult to understand why with only one character difference!

Understanding dotnet commands

When you install the .NET SDK, it includes a **command-line interface (CLI)** named `dotnet`.

Creating new projects

The .NET CLI has commands that work on the current folder to create a new project using templates:

1. On Windows, start **Command Prompt** or **Windows Terminal**. On macOS, start **Terminal**. If you are using Visual Studio Code, then you can use the integrated terminal.
2. Enter the `dotnet new list (.NET 7)`, or `dotnet new --list` or `dotnet new -l (.NET 6)` command to list your currently installed templates, as shown in *Figure 7.2*:

Template Name	Short Name	Language	Tags
Console Application	console	[C#], F#, VB	Common/Console
Class Library	classlib	[C#], F#, VB	Common/Library
WPF Application	wpf	[C#], VB	Common/WPF
WPF Class Library	wpfclasslib	[C#], VB	Common/WPF
WPF Custom Control Library	wpfcustomcontrollib	[C#], VB	Common/WPF
WPF User Control Library	wpfusercontrollib	[C#], VB	Common/WPF
Windows Forms App	winforms	[C#], VB	Common/WinForms
Windows Forms Control Library	winformscontrollib	[C#], VB	Common/WinForms
Windows Forms Class Library	winformslib	[C#], VB	Common/WinForms
Worker Service	worker	[C#], F#	Common/Worker/Web
MSTest Test Project	mstest	[C#], F#, VB	Test/MSTest
NUnit 3 Test Item	nunit-test	[C#], F#, VB	Test/NUnit
xUnit 3 Test Project	nunit	[C#], F#, VB	Test/xUnit
Razor Component	razorcomponent	[C#]	Web/ASP.NET
Razor Page	page	[C#]	Web/ASP.NET
MVC ViewImports	viewimports	[C#]	Web/ASP.NET
MVC ViewStart	viewstart	[C#]	Web/ASP.NET
Blazor Server App	blazorserver	[C#]	Web/Blazor
Blazor WebAssembly App	blazorwasm	[C#]	Web/Blazor/WebAssembly
ASP.NET Core Empty	web	[C#], F#	Web/Empty
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F#	Web/MVC
ASP.NET Core Web App	webapp	[C#]	Web/MVC/Razor Pages
ASP.NET Core with Angular	angular	[C#]	Web/MVC/SPA
ASP.NET Core with React.js	react	[C#]	Web/MVC/SPA
ASP.NET Core with React.js and Redux	reactredux	[C#]	Web/MVC/SPA
Razor Class Library	razorclasslib	[C#]	Web/Razor/Library
ASP.NET Core Web API	webapi	[C#], F#	Web/WebAPI
ASP.NET Core gRPC Service	grpc	[C#]	Web/gRPC
dotnet gitignore file	gitignore		Config
global.json file	globaljson		Config
NuGet Config	nugetconfig		Config
Dotnet local tool manifest file	tool-manifest		Config
Web Config	webconfig		Config
Solution File	sln		Solution
Protocol Buffer File	proto		Web/gRPC

Figure 7.2: A list of installed dotnet new project templates

Most dotnet command-line switches have a long and a short version, for example, `--list` or `-l`. The short ones are quicker to type but more likely to be misinterpreted by you or other humans. Sometimes more typing is clearer.

Getting information about .NET and its environment

It is useful to see what .NET SDKs and runtimes are currently installed, alongside information about the operating system, as shown in the following command:

```
dotnet --info
```

Note the results, as shown in the following partial output:

```
.NET SDK (reflecting any global.json):  
  Version: 7.0.100  
  Commit: 129d2465c8  
  
  Runtime Environment:  
    OS Name: Windows  
    OS Version: 10.0.22000  
    OS Platform: Windows  
    RID: win10-x64  
    Base Path: C:\Program Files\dotnet\sdk\7.0.100\  
  
  Host (useful for support):  
    Version: 7.0.0  
    Commit: 405337939c  
  
.NET SDKs installed:  
  3.1.416 [C:\Program Files\dotnet\sdk]  
  5.0.405 [C:\Program Files\dotnet\sdk]  
  6.0.200 [C:\Program Files\dotnet\sdk]  
  7.0.100 [C:\Program Files\dotnet\sdk]  
  
.NET runtimes installed:  
  Microsoft.AspNetCore.App 3.1.22 [...\dotnet\shared\Microsoft.AspNetCore.All]  
...
```

Managing projects

The .NET CLI has the following commands that work on the project in the current folder, to manage the project:

- `dotnet help`: Shows the command line help.
- `dotnet new`: Create a new .NET project or file.
- `dotnet tool`: Install or manage tools that extend the .NET experience.
- `dotnet workload`: Manage optional workloads like .NET MAUI.
- `dotnet restore`: This downloads dependencies for the project.
- `dotnet build`: This builds, aka compiles, a .NET project.
- `dotnet build-server`: Interact with servers started by a build.
- `dotnet msbuild`: This runs MS Build Engine commands.
- `dotnet clean`: This removes the temporary outputs from a build.
- `dotnet test`: This builds and then runs unit tests for the project.
- `dotnet run`: This builds and then runs the project.
- `dotnet pack`: This creates a NuGet package for the project.

- `dotnet publish`: This builds and then publishes the project, either with dependencies or as a self-contained application.
- `dotnet add`: This adds a reference to a package or class library to the project.
- `dotnet remove`: This removes a reference to a package or class library from the project.
- `dotnet list`: This lists the package or class library references for the project.

Publishing a self-contained app

Now that you have seen some example `dotnet` tool commands, we can publish our cross-platform console app:

1. At the command line, make sure that you are in the `DotNetEverywhere` folder.
2. Enter a command to build and publish the self-contained release version of the console application for Windows 10, as shown in the following command:

```
dotnet publish -c Release -r win10-x64 --self-contained
```

3. Note the build engine restores any needed packages, compiles the project source code into an assembly DLL, and creates a publish folder, as shown in the following output:

```
MSBuild version 17.4.0+14c24b2d3 for .NET
Determining projects to restore...
All projects are up-to-date for restore.
DotNetEverywhere -> C:\cs11dotnet7\Chapter07\DotNetEverywhere\bin\
Release\net7.0\win10-x64\DotNetEverywhere.dll
DotNetEverywhere -> C:\cs11dotnet7\Chapter07\DotNetEverywhere\bin\
Release\net7.0\win10-x64\publish\
```

4. Enter the following commands to build and publish the release versions for macOS and Linux variants:

```
dotnet publish -c Release -r osx-x64 --self-contained
dotnet publish -c Release -r osx.11.0-arm64 --self-contained
dotnet publish -c Release -r linux-x64 --self-contained
dotnet publish -c Release -r linux-arm64 --self-contained
```



Good Practice: You could automate these commands by using a scripting language like PowerShell and execute the script file on any operating system using the cross-platform PowerShell Core. Just create a file with the extension `.ps1` with the five commands in it. Then execute the file. Learn more about PowerShell at the following link: <https://github.com/markjprice/cs11dotnet7/tree/main/docs/powershell>.

5. Open Windows **File Explorer** or a macOS **Finder** window, navigate to `DotNetEverywhere\bin\Release\net7.0`, and note the output folders for the five operating systems.
6. In the `win10-x64` folder, select the `publish` folder, and note all the supporting assemblies like `Microsoft.CSharp.dll`.

7. Select the DotNetEverywhere executable file, and note it is 149 KB, as shown in *Figure 7.3*:

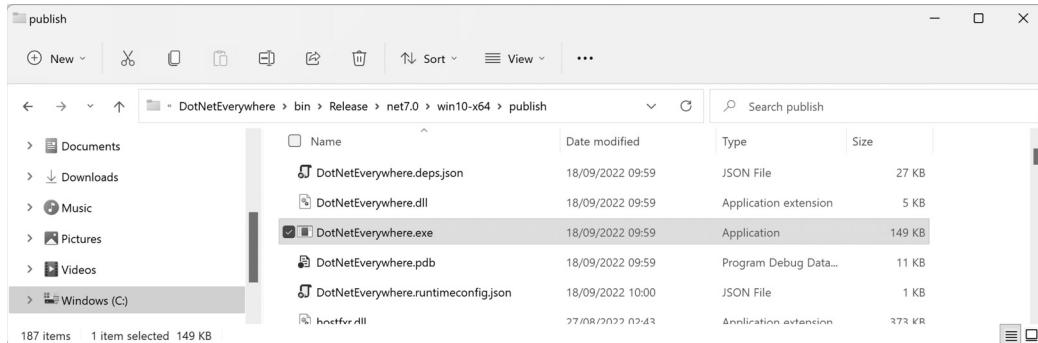


Figure 7.3: The DotNetEverywhere executable file for Windows 10 64-bit

8. If you are on Windows, then double-click to execute the program and note the result, as shown in the following output:

```
I can run everywhere!
OS Version is Microsoft Windows NT 10.0.22000.0.
I am Windows 11.
Press ENTER to stop me.
```

9. Press *Enter* to close the console app and its window.
10. Note that the total size of the publish folder and all its files is about 70 MB.
11. In the `osx.11.0-arm64` folder, select the publish folder, note all the supporting assemblies, and then select the DotNetEverywhere executable file. Note that the executable is 126 KB, and the publish folder is about 76 MB.

If you copy any of those publish folders to the appropriate operating system, the console app will run; this is because it is a self-contained deployable .NET application. For example, here it is on macOS Big Sur with Intel:

```
I can run everywhere!
OS Version is Unix 11.2.3
I am macOS.
Press ENTER to stop me.
```

This example used a console app, but you could just as easily create an ASP.NET Core website or web service, or a Windows Forms or WPF app. Of course, you can only deploy Windows desktop apps to Windows computers, not Linux or macOS.

Publishing a single-file app

To publish as a “single” file, you can specify flags when publishing. With .NET 5, single-file apps were primarily focused on Linux because there are limitations in both Windows and macOS that mean true single-file publishing is not technically possible. With .NET 6 or later, you can now create proper single-file apps on Windows.

If you can assume that .NET is already installed on the computer on which you want to run your app, then you can use the extra flags when you publish your app for release to say that it does not need to be self-contained and that you want to publish it as a single file (if possible), as shown in the following command (which must be entered on a single line):

```
dotnet publish -r win10-x64 -c Release --no-self-contained  
/p:PublishSingleFile=true
```

This will generate two files: `DotNetEverywhere.exe` and `DotNetEverywhere.pdb`. The `.exe` file is the executable. The `.pdb` file is a **program debug database** file that stores debugging information.

There is no `.exe` file extension for published applications on macOS, so if you use `osx-x64` in the command above, the filename will not have an extension.

If you prefer the `.pdb` file to be embedded in the `.exe` file, for example, to ensure it is deployed with its assembly, then add a `<DebugType>` element to the `<PropertyGroup>` element in your `.csproj` file and set it to `embedded`, as shown highlighted in the following markup:

```
<PropertyGroup>  
  
  <OutputType>Exe</OutputType>  
  <TargetFramework>net7.0</TargetFramework>  
  <Nullable>enable</Nullable>  
  <ImplicitUsings>enable</ImplicitUsings>  
  
  <RuntimeIdentifiers>  
    win10-x64;osx-x64;osx.11.0-arm64;linux-x64;linux-arm64  
  </RuntimeIdentifiers>  
  
  <DebugType>embedded</DebugType>  
  
</PropertyGroup>
```

If you cannot assume that .NET is already installed on a computer, then although Linux also only generates the two files, expect the following additional files for Windows: `coreclr.dll`, `clrjit.dll`, `clrcompression.dll`, and `mscordaccore.dll`.

Let's see an example for Windows:

1. At the command line, enter the command to build the self-contained release version of the console app for Windows 10, as shown in the following command:

```
dotnet publish -c Release -r win10-x64 --self-contained  
/p:PublishSingleFile=true
```

2. Navigate to the `DotNetEverywhere\bin\Release\net7.0\win10-x64\publish` folder and select the `DotNetEverywhere` executable file. Note that the executable is now about 64 MB, and there is also a `.pdb` file that is 11 KB. The sizes on your system will vary.

Reducing the size of apps using app trimming

One of the problems with deploying a .NET app as a self-contained app is that the .NET libraries take up a lot of space. One of the biggest needs is to reduce the size of Blazor WebAssembly components because all the .NET libraries need to be downloaded to the browser.

Luckily, you can reduce this size by not packaging unused assemblies with your deployments. Introduced with .NET Core 3.0, the app trimming system can identify the assemblies needed by your code and remove those that are not needed.

With .NET 5, the trimming went further by removing individual types, and even members like methods from within an assembly if they are not used. For example, with a Hello World console app, the `System.Console.dll` assembly is trimmed from 61.5 KB to 31.5 KB. For .NET 5, this is an experimental feature, so it is disabled by default.

With .NET 6, Microsoft added annotations to their libraries to indicate how they can be safely trimmed so the trimming of types and members was made the default. This is known as **link trim mode**.

The catch is how well the trimming identifies unused assemblies, types, and members. If your code is dynamic, perhaps using reflection, then it might not work correctly, so Microsoft also allows manual control.

Enabling assembly-level trimming

There are two ways to enable assembly-level trimming.

The first way is to add an element in the project file, as shown in the following markup:

```
<PublishTrimmed>true</PublishTrimmed>
```

The second way is to add a flag when publishing, as shown highlighted in the following command:

```
dotnet publish ... -p:PublishTrimmed=True
```

Enabling type-level and member-level trimming

There are two ways to enable type-level and member-level trimming.

The first way is to add two elements in the project file, as shown in the following markup:

```
<PublishTrimmed>true</PublishTrimmed>
<TrimMode>Link</TrimMode>
```

The second way is to add two flags when publishing, as shown highlighted in the following command:

```
dotnet publish ... -p:PublishTrimmed=True -p:TrimMode=Link
```

For .NET 6, link trim mode is the default, so you only need to specify the switch if you want to set an alternative trim mode, like `copyused`, which means assembly-level trimming.

Decompiling .NET assemblies

One of the best ways to learn how to code for .NET is to see how professionals do it.



Good Practice: You could decompile someone else's assemblies for non-learning purposes like copying their code for use in your own production library or application, but remember that you are viewing their intellectual property, so please respect that.

Decompiling using the ILSpy extension for Visual Studio 2022

For learning purposes, you can decompile any .NET assembly with a tool like ILSpy:

1. In Visual Studio 2022 for Windows, navigate to **Extensions | Manage Extensions**.
2. In the search box, enter **ilspy**.
3. For the **ILSpy 2022** extension, click **Download**.
4. Click **Close**.
5. Close Visual Studio to allow the extension to install.
6. Restart Visual Studio and reopen the **Chapter07** solution.
7. In **Solution Explorer**, right-click the **DotNetEverywhere** project and select **Open output in ILSpy**.
8. In ILSpy, in the toolbar, make sure that **C#** is selected in the drop-down list of languages to decompile into.
9. In ILSpy, in the **Assemblies** navigation tree on the left, expand **DotNetEverywhere (1.0.0.0, .NETCoreApp, v7.0)**.
10. In ILSpy, in the **Assemblies** navigation tree on the left, expand **{ }**.
11. In ILSpy, in the **Assemblies** navigation tree on the left, expand **Program**.
12. In ILSpy, in the **Assemblies** navigation tree on the left, click **<Main>\$({string[]}) : void** to show the statements in the compiler-generated **Program** class and **<Main>\$** method to reveal how interpolated strings work, as shown in *Figure 7.4*:

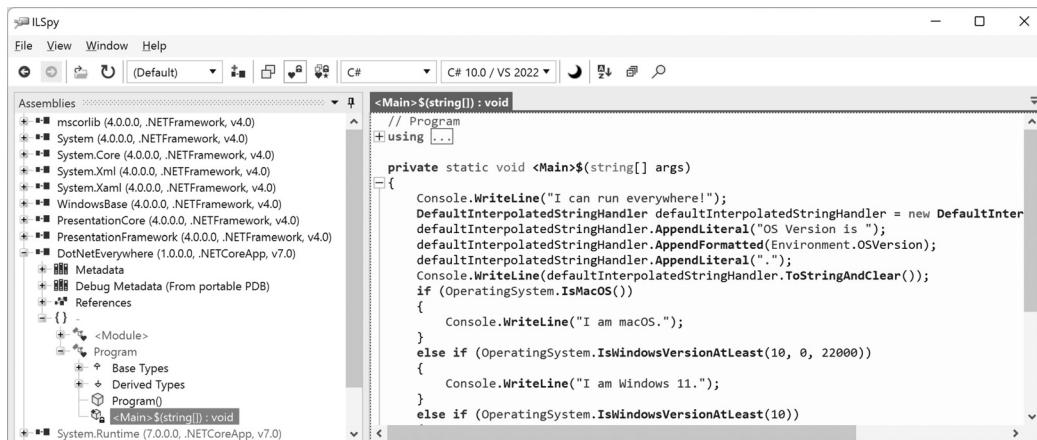


Figure 7.4: Revealing the <Main>\$ method and how interpolated strings work using ILSpy

13. In ILSpy, navigate to **File | Open....**
14. Navigate to the following folder:

```
cs11dotnet7/Chapter07/DotNetEverywhere/bin/Release/net7.0/linux-x64
```

15. Select the **System.Linq.dll** assembly and click **Open**.
16. In the **Assemblies** tree, expand the **System.Linq (7.0.0.0, .NETCoreApp, v7.0)** assembly, expand the **System.Linq** namespace, expand the **Enumerable** class, and then click the **Count<TSource>(this IEnumerable<TSource>) : int** method.
17. In the **Count** method, note the good practice of:
 - Checking the **source** parameter and throwing an **ArgumentNullException** if it is **null**.
 - Checking for interfaces that the source might implement with their own **Count** properties that would be more efficient to read.
 - The last resort of enumerating through all the items in the source and incrementing a counter, which would be the least efficient implementation.

This is shown in *Figure 7.5*:

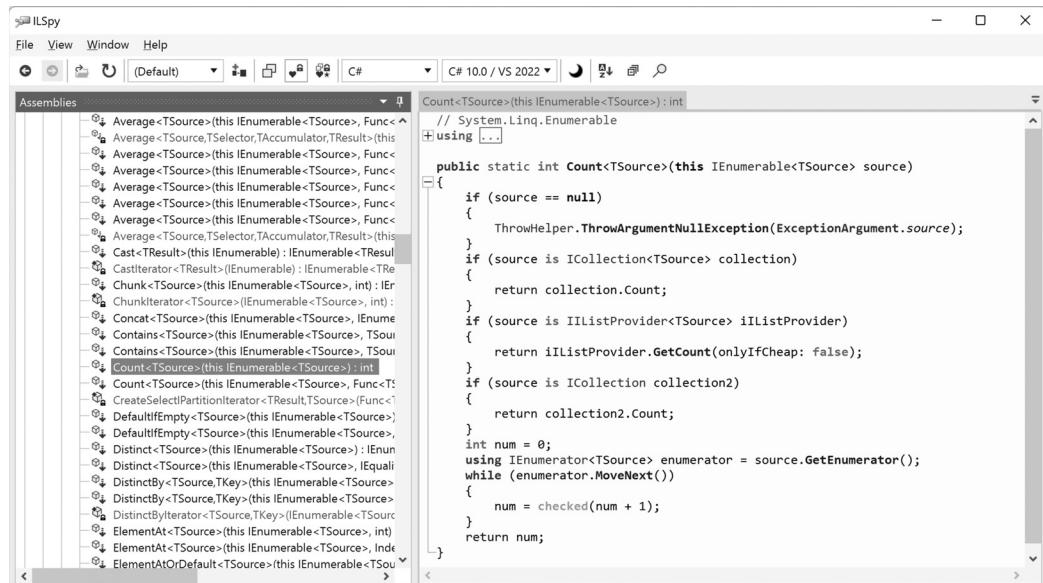


Figure 7.5: Decompiled Count method of the Enumerable class on Linux

18. Review the C# source code for the **Count** method, as shown in the following code, in preparation for reviewing the same code in **Intermediate Language (IL)**:

```
public static int Count<TSource>(this IEnumerable<TSource> source)
{
    if (source == null)
    {
        ThrowHelper.ThrowArgumentNullException(ExceptionArgument.source);
```

```
        }
        if (source is ICollection<TSource> collection)
        {
            return collection.Count;
        }
        if (source is IIListProvider<TSource> iILListProvider)
        {
            return iILListProvider.GetCount(onlyIfCheap: false);
        }
        if (source is ICollection collection2)
        {
            return collection2.Count;
        }
        int num = 0;
        using IEnumrator<TSource> enumerator = source.GetEnumrator();
        while (enumerator.MoveNext())
        {
            num = checked(num + 1);
        }
        return num;
    }
```

19. In the ILSpy toolbar, click the **Select language to decompile** dropdown and select **IL**, and then review the IL source code of the **Count** method, as shown in the following code:

```
.method public hidebysig static
    int32 Count<TSource> (
        class [System.Runtime]System.Collections.Generic.
        IEnumrable'1<!!TSource> source
    ) cil managed
{
    .custom instance void [System.Runtime]System.Runtime.CompilerServices.
    ExtensionAttribute::ctor() =
        01 00 00 00
    )
    .param type TSource
    .custom instance void System.Runtime.CompilerServices.
    NullableAttribute::ctor(uint8) =
        01 00 02 00 00
    )
    // Method begins at RVA 0x42050
    // Header size: 12
    // Code size: 103 (0x67)
    .maxstack 2
    .locals (
        [0] class [System.Runtime]System.Collections.Generic.
        ICollection'1<!!TSource>,
```

```
[1] class System.Linq.IILListProvider`1<!!TSource>,
[2] class [System.Runtime]System.Collections.ICollection,
[3] int32,
[4] class [System.Runtime]System.Collections.Generic.
IEnumerator`1<!!TSource>
)

IL_0000: ldarg.0
IL_0001: brtrue.s IL_000a

IL_0003: ldc.i4.s 16
IL_0005: call void System.Linq.
ThrowHelper::ThrowArgumentNullException(valuetype System.Linq.
ExceptionArgument)

IL_000a: ldarg.0
IL_000b: isinst class [System.Runtime]System.Collections.Generic.
ICollection`1<!!TSource>
IL_0010: stloc.0
IL_0011: ldloc.0
IL_0012: brfalse.s IL_001b

IL_0014: ldloc.0
IL_0015: callvirt instance int32 class [System.Runtime]System.
Collections.Generic.ICollection`1<!!TSource>::get_Count()
IL_001a: ret
...
IL_003e: ldc.i4.0
IL_003f: stloc.3
IL_0040: ldarg.0
IL_0041: callvirt instance class [System.Runtime]System.Collections.
Generic.IEnumerator`1<!0> class [System.Runtime]System.Collections.
Generic.IEnumerable`1<!!TSource>::GetEnumerator()
IL_0046: stloc.s 4
.try
{
    IL_0048: br.s IL_004e
    // Loop start (head: IL_004e)
    IL_004a: ldloc.3
    IL_004b: ldc.i4.1
    IL_004c: add.ovf
    IL_004d: stloc.3

    IL_004e: ldloc.s 4
    IL_0050: callvirt instance bool [System.Runtime]System.Collections.
IEnumerator::MoveNext()
    IL_0055: brtrue.s IL_004a
```

```
// end Loop

IL_0057: leave.s IL_0065
} // end .try
finally
{
    IL_0059: ldloc.s 4
    IL_005b: brfalse.s IL_0064

    IL_005d: ldloc.s 4
    IL_005f: callvirt instance void [System.Runtime]System.
IDisposable::Dispose()

    IL_0064: endfinally
} // end handler

IL_0065: ldloc.3
IL_0066: ret
} // end of method Enumerable::Count
```



Good Practice: The IL code is not especially useful unless you get very advanced with C# and .NET development, when knowing how the C# compiler translates your source code into IL code can be important. The much more useful edit windows contain the equivalent C# source code written by Microsoft experts. You can learn a lot of good practices from seeing how professionals implement types. For example, the Count method shows how to check arguments for null.

20. Close ILSpy.



You can learn how to use the ILSpy extension for Visual Studio Code at the following link:
<https://github.com/markjprice/cs11dotnet7/blob/main/docs/code-editors/vscode.md#decompiling-using-the-ilspy-extension-for-visual-studio-code>.

Viewing source links with Visual Studio 2022

Instead of decompiling, Visual Studio 2022 has a feature that allows you to view the original source code using source links. Let's see how it works:

1. Use your preferred code editor to add a new **Console App**/console project to the **Chapter07** solution/workspace named **SourceLinks**.
2. In **Program.cs**, delete the existing statements. Add statements to declare a **string** variable and then output its value and the number of characters it has, as shown in the following code:

```
string name = "Timothée Chalamet";
int length = name.Count();
```

```
Console.WriteLine($"'{name}' has {length} characters.");
```

3. Right-click in the Count method and select **Go To Implementation**.
4. Note the source code file is named **Count.cs** and it defines a partial **Enumerable** class with implementations of five count-related methods, as shown in *Figure 7.6*:

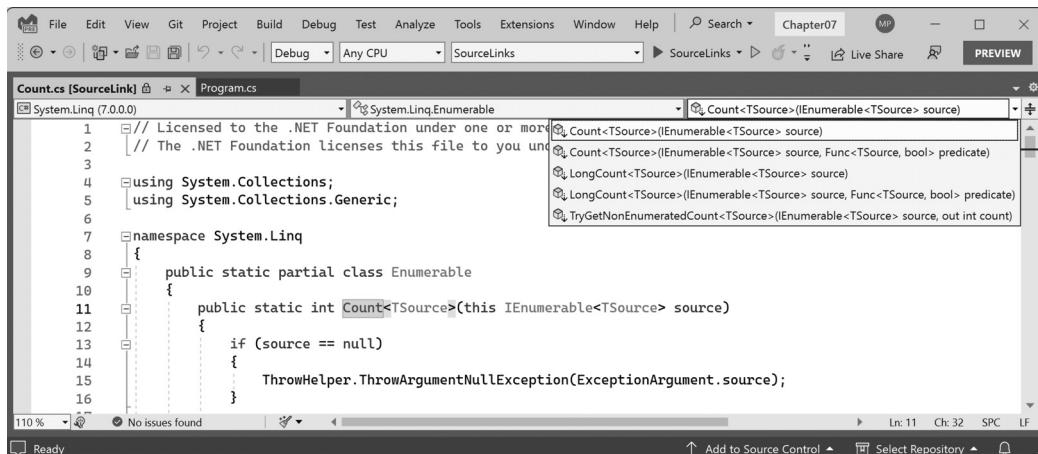


Figure 7.6: Viewing the original source file for LINQ's Count method implementation

You can learn more from viewing source links than decompiling because they show best practices for situations like how to divide up a class into partial classes for easier management. When we used the ILSpy compiler, all it could do was show all the hundreds of methods of the **Enumerable** class.



You can learn more about how source links work and how any NuGet package can support them at the following link: <https://learn.microsoft.com/en-us/dotnet/standard/library-guidance/sourcelink>.

No, you cannot technically prevent decompilation

I sometimes get asked if there is a way to protect compiled code to prevent decompilation. The quick answer is no, and if you think about it, you'll see why this must be the case. You can make it harder using obfuscation tools like **Dotfuscator**, but ultimately you cannot completely prevent it.

All compiled applications contain instructions to the platform, operating system, and hardware on which it runs. Those instructions must be functionally the same as the original source code but are just harder for a human to read. Those instructions must be readable to execute your code; they therefore must be readable to be decompiled. If you were to protect your code from decompilation using some custom technique, then you would also prevent your code from running!

Virtual machines simulate hardware and so can capture all interaction between your running application and the software and hardware that it thinks it is running on.

If you could protect your code, then you would also prevent attaching to it with a debugger and stepping through it. If the compiled application has a pdb file, then you can attach a debugger and step through the statements line-by-line. Even without the pdb file, you can still attach a debugger and get some idea of how the code works.

This is true for all programming languages. Not just .NET languages like C#, Visual Basic, and F#, but also C, C++, Delphi, and assembly language: all can be attached to for debugging or to be disassembled or decompiled. Some tools used by professionals are shown in the following table:

Type	Product	Description
Virtual Machine	VMware	Professionals like malware analysts always run software inside a VM.
Debugger	SoftICE	Runs underneath the operating system, usually in a VM.
Debugger	WinDbg	Useful for understanding Windows internals because it knows more about Windows data structures than other debuggers.
Disassembler	IDA Pro	Used by professional malware analysts.
Decompiler	HexRays	Decompiles C apps. Plugin for IDA Pro.
Decompiler	DeDe	Decompiles Delphi apps.
Decompiler	dotPeek	.NET decompiler from JetBrains.



Good Practice: Debugging, disassembling, and decompiling someone else's software is likely against its license agreement and illegal in many jurisdictions. Instead of trying to protect your intellectual property with a technical solution, the law is sometimes your only recourse.

Packaging your libraries for NuGet distribution

Before we learn how to create and package our own libraries, we will review how a project can use an existing package.

Referencing a NuGet package

Let's say that you want to add a package created by a third-party developer, for example, `Newtonsoft.Json`, a popular package for working with the **JavaScript Object Notation (JSON)** serialization format:

1. In the `AssembliesAndNamespaces` project, add a reference to the `Newtonsoft.Json` NuGet package, either using the GUI for Visual Studio 2022 or the `dotnet add package` command for Visual Studio Code.
2. Open the `AssembliesAndNamespaces.csproj` file and note that a package reference has been added, as shown in the following markup:

```
<ItemGroup>
  <PackageReference Include="newtonsoft.json" Version="13.0.1" />
</ItemGroup>
```

If you have a more recent version of the `newtonsoft.json` package, then it has been updated since this chapter was written.

Fixing dependencies

To consistently restore packages and write reliable code, it's important that you **fix dependencies**. Fixing dependencies means you are using the same family of packages released for a specific version of .NET, for example, SQLite for .NET 7.0, as shown highlighted in the following markup:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Sqlite"
      Version="7.0.0" />
  </ItemGroup>

</Project>
```

To fix dependencies, every package should have a single version with no additional qualifiers. Additional qualifiers include betas (`beta1`), release candidates (`rc4`), and wildcards (`*`).

Wildcards allow future versions to be automatically referenced and used because they always represent the most recent release. But wildcards are therefore dangerous because they could result in the use of future incompatible packages that break your code.

This can be worth the risk while writing a book where new preview versions are released every month and you do not want to keep updating the package references, as I did during 2022, and as shown in the following markup:

```
<PackageReference
  Include="Microsoft.EntityFrameworkCore.Sqlite"
  Version="7.0.0-preview.*" />
```

If you use the `dotnet add package` command, or Visual Studio's **Manage NuGet Packages**, then it will by default use the latest specific version of a package. But if you copy and paste configuration from a blog article or manually add a reference yourself, you might include wildcard qualifiers.

The following dependencies are examples of NuGet package references that are *not* fixed and therefore should be avoided unless you know the implications:

```
<PackageReference Include="System.Net.Http" Version="4.1.0-*" />
<PackageReference Include="Newtonsoft.Json" Version="13.0.2-beta1" />
```



Good Practice: Microsoft guarantees that if you fixed your dependencies to what ships with a specific version of .NET, for example, 6.0.0, those packages will all work together. Almost always fix your dependencies.

Packaging a library for NuGet

Now, let's package the SharedLibrary project that you created earlier:

1. In the SharedLibrary project, rename the `Class1.cs` file to `StringExtensions.cs`.
2. Modify its contents to provide some useful extension methods for validating various text values using regular expressions, remembering that we are targeting .NET Standard 2.0 so the compiler is C# 8.0 by default and therefore we use older syntax for namespaces and so on, as shown in the following code:

```
using System.Text.RegularExpressions;

namespace Packt.Shared
{
    public static class StringExtensions
    {
        public static bool IsValidXmlTag(this string input)
        {
            return Regex.IsMatch(input,
                @"^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$");
        }

        public static bool IsValidPassword(this string input)
        {
            // minimum of eight valid characters
            return Regex.IsMatch(input, "^[a-zA-Z0-9_-]{8,}$");
        }

        public static bool IsValidHex(this string input)
        {
            // three or six valid hex number characters
            return Regex.IsMatch(input,
                "^(#?([a-fA-F0-9]{3}|[a-fA-F0-9]{6}))$");
        }
    }
}
```



You will learn how to write regular expressions in *Chapter 8, Working with Common .NET Types*.

3. In `SharedLibrary.csproj`, modify its contents, as shown highlighted in the following markup, and note the following:

- `PackageId` must be globally unique, so you must use a different value if you want to publish this NuGet package to the <https://www.nuget.org/> public feed for others to reference and download.
- `PackageLicenseExpression` must be a value from <https://spdx.org/licenses/>, or you could specify a custom license.
- All the other elements are self-explanatory:

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

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
    <PackageId>Packt.CSdotnet.SharedLibrary</PackageId>
    <PackageVersion>7.0.0.0</PackageVersion>
    <Title>C# 11 and .NET 7 Shared Library</Title>
    <Authors>Mark J Price</Authors>
    <PackageLicenseExpression>
      MS-PL
    </PackageLicenseExpression>
    <PackageProjectUrl>
      https://github.com/markjprice/cs11dotnet7
    </PackageProjectUrl>
    <PackageIcon>packt-csdotnet-sharedlibrary.png</PackageIcon>
    <PackageRequireLicenseAcceptance>true</
    PackageRequireLicenseAcceptance>
    <PackageReleaseNotes>
      Example shared library packaged for NuGet.
    </PackageReleaseNotes>
    <Description>
      Three extension methods to validate a string value.
    </Description>
    <Copyright>
      Copyright © 2016-2022 Packt Publishing Limited
    </Copyright>
```

```
<PackageTags>string extensions packt csharp dotnet</
PackageTags>
</PropertyGroup>

<ItemGroup>
<None Include="packt-csdotnet-sharedlibrary.png">
<Pack>True</Pack>
<PackagePath></PackagePath>
</None>
</ItemGroup>

</Project>
```



Good Practice: Configuration property values that are `true` or `false` values cannot have any whitespace, so the `<PackageRequireLicenseAcceptance>` entry cannot have a carriage return and indentation as shown in the preceding markup.

4. Download the icon file and save it in the `SharedLibrary` folder from the following link: <https://github.com/markjprice/cs11dotnet7/blob/main/vs4win/Chapter07/SharedLibrary/packt-csdotnet-sharedlibrary.png>.
5. Build the release assembly:
 - In Visual Studio 2022, select **Release** in the toolbar, and then navigate to **Build | Build SharedLibrary**.
 - In Visual Studio Code, in **Terminal**, enter `dotnet build -c Release`.

If we had not set `<GeneratePackageOnBuild>` to `true` in the project file, then we would have to create a NuGet package manually using the following additional steps:

- In Visual Studio 2022, navigate to **Build | Pack SharedLibrary**.
- In Visual Studio Code, in **Terminal**, enter `dotnet pack -c Release`.

Publishing a package to a public NuGet feed

If you want everyone to be able to download and use your NuGet package, then you must upload it to a public NuGet feed like Microsoft's:

1. Start your favorite browser and navigate to the following link: <https://www.nuget.org/packages/manage/upload>.

2. You will need to sign up for, and then sign in with, a Microsoft account at <https://www.nuget.org/> if you want to upload a NuGet package for other developers to reference as a dependency package.
3. Click the **Browse...** button and select the .nupkg file that was created by generating the NuGet package. The folder path should be `cs11dotnet7\Chapter07\SharedLibrary\bin\Release` and the file is named `Packt.CSdotnet.SharedLibrary.7.0.0.nupkg`.
4. Verify that the information you entered in the `SharedLibrary.csproj` file has been correctly filled in, and then click **Submit**.
5. Wait a few seconds, and you will see a success message showing that your package has been uploaded, as shown in *Figure 7.7*:

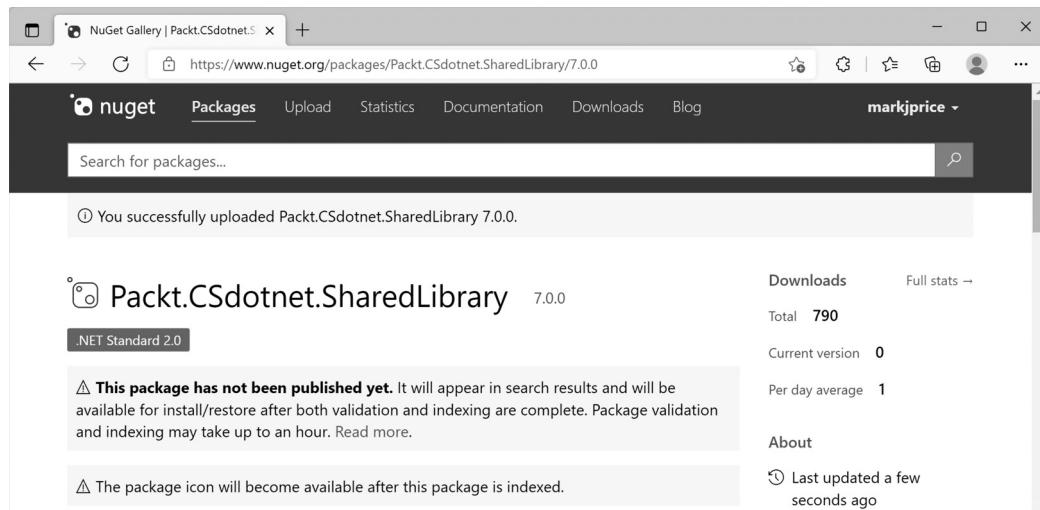


Figure 7.7: A NuGet package upload message

 **Good Practice:** If you get an error, then review the project file for mistakes, or read more information about the `PackageReference` format at <https://docs.microsoft.com/en-us/nuget/reference/msbuild-targets>.

6. Click the **Frameworks** tab, and note that because we targeted .NET Standard 2.0, our class library can be used by every .NET platform, as shown in *Figure 7.8*:

Product	Versions
.NET	net5.0, net5.0-windows, net6.0, net6.0-android, net6.0-ios, net6.0-maccatalyst, net6.0-macos, net6.0-tvos, net6.0-windows
.NET Core	netcoreapp2.0, netcoreapp2.1, netcoreapp2.2, netcoreapp3.0, netcoreapp3.1
.NET Standard	netstandard2.0, netstandard2.1
.NET Framework	net461, net462, net463, net47, net471, net472, net48

Figure 7.8: .NET Standard 2.0 class library NuGet packages can be used by all .NET platforms

Publishing a package to a private NuGet feed

Organizations can host their own private NuGet feeds. This can be a handy way for many developer teams to share work. You can read more at the following link:

<https://docs.microsoft.com/en-us/nuget/hosting-packages/overview>

Exploring NuGet packages with a tool

A handy tool named NuGet Package Explorer for opening and reviewing more details about a NuGet package was created by Uno Platform. As well as being a website, it can be installed as a cross-platform app. Let's see what it can do:

1. Start your favorite browser and navigate to the following link: <https://nuget.info>.
2. In the search box, enter `Packt.CSdotnet.SharedLibrary`.
3. Select the package `v7.0.0` published by **Mark J Price** and then click the **Open** button.
4. In the **Contents** section, expand the `lib` folder and the `netstandard2.0` folder.

5. Select `SharedLibrary.dll`, and note the details, as shown in *Figure 7.9*:

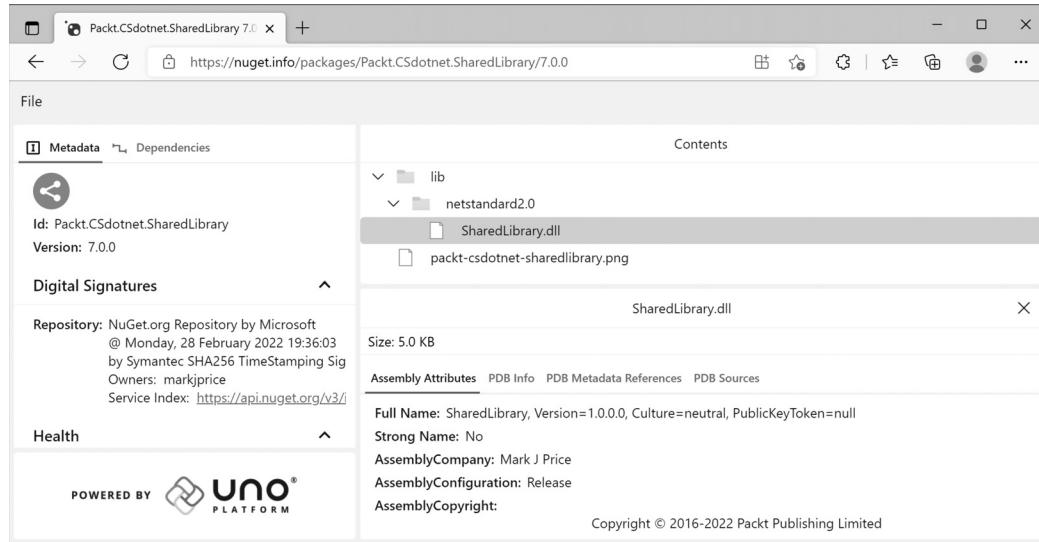


Figure 7.9: Exploring my package using NuGet Package Explorer from Uno Platform

6. If you want to use this tool locally in the future, click the install button in your browser.
7. Close your browser.

Not all browsers support installing web apps like this. I recommend Chrome for testing and development.

Testing your class library package

You will now test your uploaded package by referencing it in the `AssembliesAndNamespaces` project:

1. In the `AssembliesAndNamespaces` project, add a reference to your (or my) package, as shown highlighted in the following markup:

```
<ItemGroup>
  <PackageReference Include="newtonsoft.json" Version="13.0.1" />
  <PackageReference Include="packt.csdotnet.sharedlibrary"
    Version="7.0.0" />
</ItemGroup>
```

2. Build the `AssembliesAndNamespaces` console app.
3. In `Program.cs`, import the `Packt.Shared` namespace.
4. In `Program.cs`, prompt the user to enter some `string` values, and then validate them using the extension methods in the package, as shown in the following code:

```
Write("Enter a color value in hex: ");
string? hex = ReadLine() // or "00fffc8"
```

```
    WriteLine("Is {0} a valid color value? {1}",
      arg0: hex, arg1: hex.IsValidHex());

    Write("Enter a XML element: ");
    string? xmlTag = ReadLine(); // or "<h1 class='<\' />"
    WriteLine("Is {0} a valid XML element? {1}",
      arg0: xmlTag, arg1: xmlTag.IsValidXmlTag());

    Write("Enter a password: ");
    string? password = ReadLine(); // or "secretsauce"
    WriteLine("Is {0} a valid password? {1}",
      arg0: password, arg1: password.IsValidPassword());
```

5. Run the console app, enter some values as prompted, and view the results, as shown in the following output:

```
Enter a color value in hex: 00ffc8
Is 00ffc8 a valid color value? True
Enter an XML element: <h1 class="<" />
Is <h1 class="<" /> a valid XML element? False
Enter a password: secretsauce
Is secretsauce a valid password? True
```

Porting from .NET Framework to modern .NET

If you are an existing .NET Framework developer, then you may have existing applications that you think you should port to modern .NET. But you should carefully consider if porting is the right choice for your code, because sometimes, the best choice is not to port.

For example, you might have a complex website project that runs on .NET Framework 4.8 but is only visited by a small number of users. If it works and handles the visitor traffic on minimal hardware, then potentially spending months porting it to a modern .NET platform could be a waste of time. But if the website currently requires many expensive Windows servers, then the cost of porting could eventually pay off if you can migrate to fewer, less costly Linux servers.

Could you port?

Modern .NET has great support for the following types of applications on Windows, macOS, and Linux, so they are good candidates for porting:

- **ASP.NET Core** websites, including Razor Pages and MVC
- **ASP.NET Core** web services (REST/HTTP), including **Web APIs**, **Minimal APIs**, and **ODATA**
- **ASP.NET Core**-hosted services, including **gRPC**, **GraphQL**, and **SignalR**
- **Console App** command-line interfaces

Modern .NET has decent support for the following types of applications on Windows, so they are potential candidates for porting:

- **Windows Forms** applications
- **Windows Presentation Foundation (WPF)** applications

Modern .NET has good support for the following types of applications on cross-platform desktop and mobile devices:

- **Xamarin** apps for mobile iOS and Android
- **.NET MAUI** for desktop Windows and macOS, or mobile iOS and Android

Modern .NET does not support the following types of legacy Microsoft projects:

- **ASP.NET Web Forms** websites. These might be best reimplemented using **ASP.NET Core Razor Pages** or **Blazor**.
- **Windows Communication Foundation (WCF)** services (but there is an open-source project named **CoreWCF** that you might be able to use depending on requirements). WCF services might be better reimplemented using **ASP.NET Core gRPC** services.
- **Silverlight** applications. These might be best reimplemented using **Blazor** or **.NET MAUI**.

Silverlight and ASP.NET Web Forms applications will never be able to be ported to modern .NET, but existing Windows Forms and WPF applications could be ported to .NET on Windows to benefit from the new APIs and faster performance.

Legacy ASP.NET MVC web applications and ASP.NET Web API web services currently on .NET Framework could be ported to modern .NET and then be hosted on Windows, Linux, or macOS.

Should you port?

Even if you *could* port, *should* you? What benefits do you gain? Some common benefits include the following:

- **Deployment to Linux, Docker, or Kubernetes for websites and web services:** These OSes are lightweight and cost-effective as website and web service platforms, especially when compared to the more costly Windows Server.
- **Removal of dependency on IIS and System.Web.dll:** Even if you continue to deploy to Windows Server, ASP.NET Core can be hosted on lightweight, higher-performance Kestrel (or other) web servers.
- **Command-line tools:** Tools that developers and administrators use to automate their tasks are often built as console applications. The ability to run a single tool cross-platform is very useful.

Differences between .NET Framework and modern .NET

There are three key differences, as shown in the following table:

Modern .NET	.NET Framework
Distributed as NuGet packages, so each application can be deployed with its own app-local copy of the version of .NET that it needs.	Distributed as a system-wide, shared set of assemblies (literally, in the Global Assembly Cache (GAC)).
Split into small, layered components, so a minimal deployment can be performed.	Single, monolithic deployment.
Removes older technologies, such as ASP.NET Web Forms, and non-cross-platform features, such as AppDomains, .NET Remoting, and binary serialization.	As well as some similar technologies to those in modern .NET like ASP.NET Core MVC, it also retains some older technologies, such as ASP.NET Web Forms.

.NET Portability Analyzer

Microsoft has a useful tool that you can run against your existing applications to generate a report for porting. You can watch a demonstration of the tool at the following link: <https://learn.microsoft.com/en-us/shows/seth-juarez/brief-look-net-portability-analyzer>.

.NET Upgrade Assistant

Microsoft's latest tool for upgrading legacy projects to modern .NET is the **.NET Upgrade Assistant**.

For my day job, I used to work for a company named Optimizely. They have an enterprise-scale **Digital Experience Platform (DXP)** based on .NET comprising a **Content Management System (CMS)** and a Digital Commerce platform. Microsoft needed a challenging migration project to design and test the .NET Upgrade Assistant with, so we worked with them to build a great tool.

Currently, it supports the following .NET Framework project types and more will be added later:

- ASP.NET MVC
- Windows Forms
- WPF
- Console Application
- Class Library

It is installed as a global dotnet tool, as shown in the following command:

```
dotnet tool install -g upgrade-assistant
```

You can read more about this tool and how to use it at the following link:

<https://docs.microsoft.com/en-us/dotnet/core/porting/upgrade-assistant-overview>

Using non-.NET Standard libraries

Most existing NuGet packages can be used with modern .NET, even if they are not compiled for .NET Standard or a modern version like .NET 7. If you find a package that does not officially support .NET Standard, as shown on its nuget.org web page, you do not have to give up. You should try it and see if it works.

For example, there is a package of custom collections for handling matrices created by Dialect Software LLC, documented at the following link:

<https://www.nuget.org/packages/DialectSoftware.Collections.Matrix/>

This package was last updated in 2013, which was long before .NET Core or .NET 7 existed, so this package was built for .NET Framework. If an assembly package like this only uses APIs available in .NET Standard, it can be used in a modern .NET project.

Let's try using it and see if it works:

1. In the `AssembliesAndNamespaces` project, add a package reference for Dialect Software's package, as shown in the following markup:

```
<PackageReference
  Include="dialectsoftware.collections.matrix"
  Version="1.0.0" />
```

2. Build the `AssembliesAndNamespaces` project to restore packages.
3. In `Program.cs`, add statements to import the `DialectSoftware.Collections` and `DialectSoftware.Collections.Generics` namespaces.
4. Add statements to create instances of `Axis` and `Matrix<T>`, populate them with values, and output them, as shown in the following code:

```
Axis x = new("x", 0, 10, 1);
Axis y = new("y", 0, 4, 1);
Matrix<long> matrix = new(new[] { x, y });

for (int i = 0; i < matrix.Axes[0].Points.Length; i++)
{
    matrix.Axes[0].Points[i].Label = "x" + i.ToString();
}

for (int i = 0; i < matrix.Axes[1].Points.Length; i++)
{
    matrix.Axes[1].Points[i].Label = "y" + i.ToString();
}

foreach (long[] c in matrix)
{
    matrix[c] = c[0] + c[1];
}
```

```
foreach (long[] c in matrix)
{
    WriteLine("{0},{1} ({2},{3}) = {4}",
        matrix.Axes[0].Points[c[0]].Label,
        matrix.Axes[1].Points[c[1]].Label,
        c[0], c[1], matrix[c]);
}
```

5. Run the code, noting the warning message and the results, as shown in the following output:

```
warning NU1701: Package 'DialectSoftware.Collections.Matrix
1.0.0' was restored using '.NETFramework,Version=v4.6.1,
.NETFramework,Version=v4.6.2, .NETFramework,Version=v4.7,
.NETFramework,Version=v4.7.1, .NETFramework,Version=v4.7.2,
.NETFramework,Version=v4.8' instead of the project target framework
'net7.0'. This package may not be fully compatible with your project.
x0,y0 (0,0) = 0
x0,y1 (0,1) = 1
x0,y2 (0,2) = 2
x0,y3 (0,3) = 3
...
```

Even though this package was created before modern .NET existed, and the compiler and runtime have no way of knowing if it will work and therefore show warnings, because it happens to only call .NET Standard-compatible APIs, it works.

Working with preview features

It is a challenge for Microsoft to deliver some new features that have cross-cutting effects across many parts of .NET like the runtime, language compilers, and API libraries. It is the classic chicken and egg problem. What do you do first?

From a practical perspective, it means that although Microsoft might have completed most of the work needed for a feature, the whole thing might not be ready until very late in their now annual cycle of .NET releases, too late for proper testing in “the wild.”

So, from .NET 6 onward, Microsoft will include preview features in **general availability (GA)** releases. Developers can opt into these preview features and provide Microsoft with feedback. In a later GA release, they can be enabled for everyone.



It is important to note that this topic is about *preview features*. This is different from a preview version of .NET or a preview version of Visual Studio 2022. Microsoft releases preview versions of Visual Studio and .NET while developing them to get feedback from developers and then do a final GA release. At GA, the feature is available for everyone. Before GA, the only way to get the new functionality is to install a preview version. *Preview features* are different because they are installed with GA releases and must be optionally enabled.

For example, when Microsoft released .NET SDK 6.0.200 in February 2022, it included the C# 11 compiler as a preview feature. This meant that .NET 6 developers could optionally set the language version to preview, and then start exploring C# 11 features like raw string literals and the required keyword.



Good Practice: Preview features are not supported in production code. Preview features are likely to have breaking changes before the final release. Enable preview features at your own risk.

Requiring preview features

The `[RequiresPreviewFeatures]` attribute is used to indicate assemblies, types, or members that use and therefore require warnings about preview features. A code analyzer then scans for this assembly and generates warnings if needed. If your code does not use any preview features, you will not see any warnings. If you use any preview features, then your code should warn consumers of your code that you use preview features.

Enabling preview features

In the project file, add an `EnablePreviewFeatures` element to enable preview language features, as shown highlighted in the following markup:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <EnablePreviewFeatures>true</EnablePreviewFeatures>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>

</Project>
```

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring with deeper research into topics of this chapter.

Exercise 7.1 – Test your knowledge

Answer the following questions:

1. What is the difference between a namespace and an assembly?
2. How do you reference another project in a `.csproj` file?

3. What is the benefit of a tool like ILSpy?
4. Which .NET type does the C# `float` alias represent?
5. When porting an application from .NET Framework to modern .NET, what tool should you run before porting, and what tool could you run to perform much of the porting work?
6. What is the difference between framework-dependent and self-contained deployments of .NET applications?
7. What is a RID?
8. What is the difference between the `dotnet pack` and `dotnet publish` commands?
9. What types of applications written for .NET Framework can be ported to modern .NET?
10. Can you use packages written for .NET Framework with modern .NET?

Exercise 7.2 – Explore topics

Use the links on the following page to learn more detail about the topics covered in this chapter:

<https://github.com/markjprice/cs11dotnet7/blob/main/book-links.md#chapter-7---packaging-and-distributing-net-types>

Exercise 7.3 – Explore PowerShell

PowerShell is Microsoft's scripting language for automating tasks on every operating system. Microsoft recommends Visual Studio Code with the PowerShell extension for writing PowerShell scripts.

Since PowerShell is its own extensive language, there is not enough space in this book to cover it. Instead, I have created some supplementary pages on the book's GitHub repository to introduce you to some key concepts and show some examples:

<https://github.com/markjprice/cs11dotnet7/tree/main/docs/powershell>

Summary

In this chapter, we:

- Reviewed the journey to .NET 7 for Base Class Library functionality.
- Explored the relationship between assemblies and namespaces.
- Learned how to decompile .NET assemblies for educational purposes.
- Saw options for publishing an app for distribution to multiple operating systems.
- Packaged and distributed a class library.
- Discussed options for porting existing .NET Framework code bases.
- Learned how to activate preview features.

In the next chapter, you will learn about some common Base Class Library types that are included with modern .NET.

12

Introducing Web Development Using ASP.NET Core

The third and final part of this book is about web development using ASP.NET Core. You will learn how to build cross-platform projects such as websites, web services, and web browser apps.

Microsoft calls platforms for building applications **app models** or **workloads**.

I recommend that you work through this and subsequent chapters sequentially because later chapters will reference projects in earlier chapters, and you will build up sufficient knowledge and skills to tackle the trickier problems in later chapters.

In this chapter, we will cover the following topics:

- Understanding ASP.NET Core
- New features in ASP.NET Core
- Structuring projects
- Building an entity model for use in the rest of the book
- Understanding web development

Understanding ASP.NET Core

Since this book is about C# and .NET, we will learn about app models that use them to build the practical applications that we will encounter in the remaining chapters of this book.



Learn More: Microsoft has extensive guidance for implementing app models in its .NET Application Architecture Guidance documentation, which you can read at the following link: <https://www.microsoft.com/net/learn/architecture>.

Microsoft ASP.NET Core is part of a history of Microsoft technologies used to build websites and services that have evolved over the years:

- **Active Server Pages (ASP)** was released in 1996 and was Microsoft's first attempt at a platform for dynamic server-side execution of website code. ASP files contain a mix of HTML and code that executes on the server written in the VBScript language.
- **ASP.NET Web Forms** was released in 2002 with the .NET Framework and was designed to enable non-web developers, such as those familiar with Visual Basic, to quickly create websites by dragging and dropping visual components and writing event-driven code in Visual Basic or C#. Web Forms should be avoided for new .NET Framework web projects in favor of ASP.NET MVC.
- **Windows Communication Foundation (WCF)** was released in 2006 and enables developers to build SOAP and REST services. SOAP is powerful but complex, so it should be avoided unless you need advanced features, such as distributed transactions and complex messaging topologies.
- **ASP.NET MVC** was released in 2009 to cleanly separate the concerns of web developers between the **models**, which temporarily store the data; the **views**, which present the data using various formats in the UI; and the **controllers**, which fetch the model and pass it to a view. This separation enables improved reuse and unit testing.
- **ASP.NET Web API** was released in 2012 and enables developers to create HTTP services (aka REST services) that are simpler and more scalable than SOAP services.
- **ASP.NET SignalR** was released in 2013 and enables real-time communication in websites by abstracting underlying technologies and techniques, such as WebSockets and long polling. This enables website features such as live chat or updates to time-sensitive data such as stock prices across a wide variety of web browsers, even when they do not support an underlying technology such as WebSockets.
- **ASP.NET Core** was released in 2016 and combines modern implementations of .NET Framework technologies such as MVC, Web API, and SignalR, with newer technologies such as Razor Pages, gRPC, and Blazor, all running on modern .NET. Therefore, it can execute cross-platform. ASP.NET Core has many project templates to get you started with its supported technologies.



Good Practice: Choose ASP.NET Core to develop websites and web services because it includes web-related technologies that are modern and cross-platform.

Classic ASP.NET versus modern ASP.NET Core

Until modern .NET, ASP.NET was built on top of a large assembly in .NET Framework named **System.Web.dll** and it was tightly coupled to Microsoft's Windows-only web server named **Internet Information Services (IIS)**. Over the years, this assembly has accumulated a lot of features, many of which are not suitable for modern cross-platform development.

ASP.NET Core is a major redesign of ASP.NET. It removes the dependency on the **System.Web.dll** assembly and IIS and is composed of modular lightweight packages, just like the rest of modern .NET. Using IIS as the web server is still supported by ASP.NET Core, but there is a better option.

You can develop and run ASP.NET Core applications cross-platform on Windows, macOS, and Linux. Microsoft has even created a cross-platform, super-performant web server named **Kestrel**, and the entire stack is open source.

ASP.NET Core 2.2 or later projects default to the new in-process hosting model. This gives a 400% performance improvement when hosting in Microsoft IIS, but Microsoft still recommends using Kestrel for even better performance.

Building websites using ASP.NET Core

Websites are made up of multiple web pages loaded statically from the filesystem or generated dynamically by a server-side technology such as ASP.NET Core. A web browser makes **GET** requests using **Uniform Resource Locators (URLs)** that identify each page and can manipulate data stored on the server using **POST**, **PUT**, and **DELETE** requests.

With many websites, the web browser is treated as a presentation layer, with almost all the processing performed on the server side. Some JavaScript might be used on the client side to implement form validation warnings and some presentation features, such as carousels.

ASP.NET Core provides multiple technologies for building websites:

- **ASP.NET Core Razor Pages** and **Razor class libraries** are ways to dynamically generate HTML for simple websites. You will learn about them in detail in *Chapter 13, Building Websites Using ASP.NET Core Razor Pages*.
- **ASP.NET Core MVC** is an implementation of the **Model-View-Controller (MVC)** design pattern that is popular for developing complex websites. You will learn about it in detail in *Chapter 14, Building Websites Using the Model-View-Controller Pattern*.
- **Blazor** lets you build user interface components using C# and .NET instead of a JavaScript-based UI framework like Angular, React, and Vue. **Blazor WebAssembly** runs your code in the browser like a JavaScript-based framework would. **Blazor Server** runs your code on the server and updates the web page dynamically. You will learn about Blazor in detail in *Chapter 16, Building User Interfaces Using Blazor*. Blazor is not just for building websites; it can also be used to create hybrid mobile and desktop apps by being hosted inside a .NET MAUI app.

Building websites using a content management system

Most websites have a lot of content, and if developers had to be involved every time some content needed to be changed, that would not scale well. A **Content Management System (CMS)** enables developers to define content structure and templates to provide consistency and good design while making it easy for a non-technical content owner to manage the actual content. They can create new pages or blocks of content, and update existing content, knowing it will look great for visitors with minimal effort.

There is a multitude of CMSs available for all web platforms, like WordPress for PHP or Django CMS for Python. CMSs that support modern .NET include Optimizely Content Cloud, Piranha CMS, and Orchard Core.

The key benefit of using a CMS is that it provides a friendly content management user interface. Content owners log in to the website and manage the content themselves. The content is then rendered and returned to visitors using ASP.NET Core MVC controllers and views, or via web service endpoints, known as a **headless CMS**, to provide that content to “heads” implemented as mobile or desktop apps, in-store touchpoints, or clients built with JavaScript frameworks or Blazor.

This book does not cover .NET CMSs, so I have included links where you can learn more about them in the GitHub repository:

<https://github.com/markjprice/cs11dotnet7/blob/main/book-links.md#net-content-management-systems>

Building web applications using SPA frameworks

Web applications are often built using technologies known as **Single-Page Applications (SPAs)** frameworks, such as Blazor WebAssembly, Angular, React, Vue, or a proprietary JavaScript library. They can make requests to a backend web service for getting more data when needed and posting updated data using common serialization formats such as XML and JSON. The canonical examples are Google web apps like Gmail, Maps, and Docs.

With a web application, the client side uses JavaScript frameworks or Blazor WebAssembly to implement sophisticated user interactions, but most of the important processing and data access still happens on the server side, because the web browser has limited access to local system resources.

JavaScript is loosely typed and is not designed for complex projects, so most JavaScript libraries these days use TypeScript, which adds strong typing to JavaScript and is designed with many modern language features for handling complex implementations.

.NET SDK has project templates for JavaScript and TypeScript-based SPAs, but we will not spend any time learning how to build JavaScript- and TypeScript-based SPAs in this book, even though they are commonly used with ASP.NET Core as the backend, because this book is about C#; it is not about other languages.

In summary, C# and .NET can be used on both the server side and the client side to build websites, as shown in *Figure 12.1*:

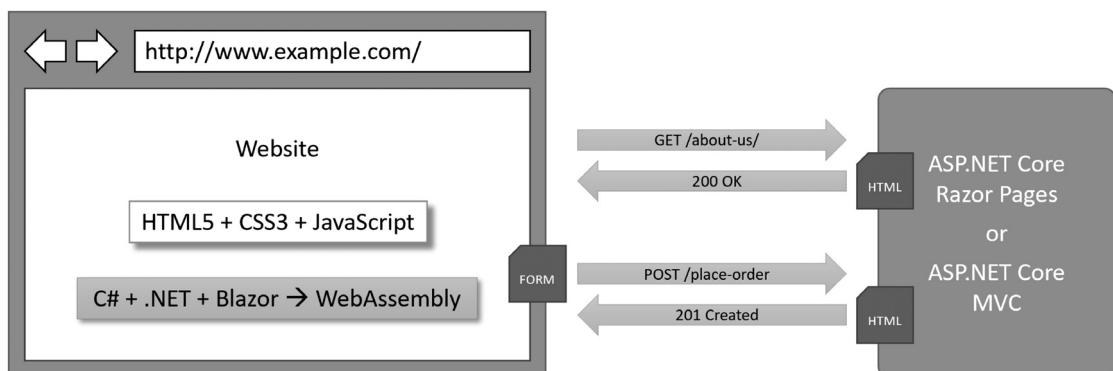


Figure 12.1: The use of C# and .NET to build websites on both the server side and the client side

Building web and other services

Although we will not learn about JavaScript- and TypeScript-based SPAs, we will learn how to build a web service using the **ASP.NET Core Web API**, and then call that web service from the server-side code in our ASP.NET Core websites. Later, we will call that web service from Blazor WebAssembly components and cross-platform mobile and desktop apps.

There are no formal definitions, but services are sometimes described based on their complexity:

- **Service:** All functionality needed by a client app in one monolithic service.
- **Microservice:** Multiple services that each focus on a smaller set of functionalities.
- **Nanoservice:** A single function provided as a service. Unlike services and microservices that are hosted 24/7/365, nanoservices are often inactive until called upon to reduce resources and costs.

New features in ASP.NET Core

Over the past few years, Microsoft has rapidly expanded the capabilities of ASP.NET Core. You should note which .NET platforms are supported, as shown in the following list:

- ASP.NET Core 1.0 to 2.2 runs on either .NET Core or .NET Framework.
- ASP.NET Core 3.0 or later only runs on .NET Core 3.0 or later.

ASP.NET Core 1.0

ASP.NET Core 1.0 was released in June 2016 and focused on implementing a minimum API suitable for building modern cross-platform web apps and services for Windows, macOS, and Linux.

ASP.NET Core 1.1

ASP.NET Core 1.1 was released in November 2016 and focused on bug fixes and general improvements to features and performance.

ASP.NET Core 2.0

ASP.NET Core 2.0 was released in August 2017 and focused on adding new features such as Razor Pages, bundling assemblies into a `Microsoft.AspNetCore.App` metapackage, targeting .NET Standard 2.0, providing a new authentication model and performance improvements.

The biggest new features introduced with ASP.NET Core 2.0 are ASP.NET Core Razor Pages, which is covered in *Chapter 13, Building Websites Using ASP.NET Core Razor Pages*, and ASP.NET Core OData support. OData is covered in my other book, *Apps and Services with .NET 7*.

ASP.NET Core 2.1

ASP.NET Core 2.1 was released in May 2018 and was a **Long Term Support (LTS)** release, meaning it was supported for three years until August 21, 2021 (LTS designation was not officially assigned to it until August 2018 with version 2.1.3).

It focused on adding new features such as SignalR for real-time communication, Razor class libraries for reusing web components, ASP.NET Core Identity for authentication, and better support for HTTPS and the European Union's General Data Protection Regulation (GDPR), including the topics listed in the following table:

Feature	Chapter	Topic
Razor class libraries	13	Using Razor class libraries
GDPR support	14	Creating and exploring an ASP.NET Core MVC website
Identity UI library and scaffolding	14	Exploring an ASP.NET Core MVC website
Integration tests	14	Testing an ASP.NET Core MVC website
[ApiController], ActionResult<T>	15	Creating an ASP.NET Core Web API project
Problem details	15	Implementing a Web API controller
IHttpClientFactory	15	Configuring HTTP clients using HttpClientFactory

ASP.NET Core 2.2

ASP.NET Core 2.2 was released in December 2018 and focused on improving the building of RESTful HTTP APIs, updating the project templates to Bootstrap 4 and Angular 6, an optimized configuration for hosting in Azure, and performance improvements, including the topics listed in the following table:

Feature	Chapter	Topic
HTTP/2 in Kestrel	13	Classic ASP.NET versus modern ASP.NET Core
In-process hosting model	13	Creating an ASP.NET Core project
Endpoint routing	13	Understanding endpoint routing
Health Checks Middleware	15	Implementing health checks
Open API analyzers	15	Implementing Open API analyzers and conventions

ASP.NET Core 3.0

ASP.NET Core 3.0 was released in September 2019 and focused on fully leveraging .NET Core 3.0 and .NET Standard 2.1, which meant it could not support .NET Framework, and it added useful refinements, including the topics listed in the following table:

Feature	Chapter	Topic
Static assets in Razor class libraries	13	Using Razor class libraries
New options for MVC service registration	14	Understanding ASP.NET Core MVC startup
Blazor Server	16	Building components using Blazor Server

ASP.NET Core 3.1

ASP.NET Core 3.1 was released in December 2019 and is an LTS release, meaning it will be supported until December 13, 2022. It focused on refinements like partial class support for Razor components and a new `<component>` tag helper.

Blazor WebAssembly 3.2

Blazor WebAssembly 3.2 was released in May 2020. It was a Current (now known as Standard) release, meaning that projects had to be upgraded to the .NET 5 version within three months of the .NET 5 release, that is, by February 10, 2021. Microsoft finally delivered on the promise of full-stack web development with .NET, and both Blazor Server and Blazor WebAssembly are covered in *Chapter 16, Building User Interfaces Using Blazor*.

ASP.NET Core 5.0

ASP.NET Core 5.0 was released in November 2020 and focused on bug fixes, performance improvements, using caching for certificate authentication, HPACK dynamic compression of HTTP/2 response headers in Kestrel, nullable annotations for ASP.NET Core assemblies, and a reduction in container image sizes, including the topics listed in the following table:

Feature	Chapter	Topic
Extension method to allow anonymous access to an endpoint	15	Securing web services
JSON extension methods for <code>HttpRequest</code> and <code>HttpResponse</code>	15	Getting customers as JSON in the controller

ASP.NET Core 6.0

ASP.NET Core 6.0 was released in November 2021 and focused on productivity improvements like minimizing code to implement basic websites and services, support for .NET Hot Reload, and new hosting options for Blazor, like hybrid apps using .NET MAUI, including the topics listed in the following table:

Feature	Chapter	Topic
New empty web project template	13	Understanding the empty web template
Minimal APIs	15	Implementing Minimal Web APIs
Blazor WebAssembly AOT	16	Enabling Blazor WebAssembly ahead-of-time compilation

ASP.NET Core 7.0

ASP.NET Core 7.0 was released in November 2022 and focused on filling well-known gaps in functionality like HTTP/3 support, output caching, and many quality-of-life improvements to Blazor, including the topics listed in the following table:

Feature	Chapter	Topic
HTTP request decompression	13	Enabling request decompression support
HTTP/3 support	13	Enabling HTTP/3 support
Output caching	14	Using a filter to cache output
W3C log additional headers	15	Support for logging additional request headers in W3CLogger
HTTP/3 client support	15	Enabling HTTP/3 support for HttpClient
Blazor Empty templates	16	Comparing Blazor project templates
Location change support	16	Enabling location change event handling



You can read the full ASP.NET Core Roadmap for .NET 7 at the following link: <https://github.com/dotnet/aspnetcore/issues/39504>.

Structuring projects

How should you structure your projects? So far, we have built small individual console apps to illustrate language or library features. In the rest of this book, we will build multiple projects using different technologies that work together to provide a single solution.

With large, complex solutions, it can be difficult to navigate through all the code. So, the primary reason to structure your projects is to make it easier to find components. It is good to have an overall name for your solution or workspace that reflects the application or solution.

We will build multiple projects for a fictional company named **Northwind**. We will name the solution or workspace **PracticalApps** and use the name **Northwind** as a prefix for all the project names.

There are many ways to structure and name projects and solutions, for example, using a folder hierarchy as well as a naming convention. If you work in a team, make sure you know how your team does it.

Structuring projects in a solution or workspace

It is good to have a naming convention for your projects in a solution or workspace so that any developer can tell what each one does instantly. A common choice is to use the type of project, for example, class library, console app, website, and so on, as shown in the following table:

Name	Description
Northwind.Common	A class library project for common types like interfaces, enums, classes, records, and structs, used across multiple projects.
Northwind.Common.EntityModels	A class library project for common EF Core entity models. Entity models are often used on both the server and client side, so it is best to separate dependencies on specific database providers.
Northwind.Common.DataContext	A class library project for the EF Core database context with dependencies on specific database providers.
Northwind.Web	An ASP.NET Core project for a simple website that uses a mixture of static HTML files and dynamic Razor Pages.
Northwind.Razor.Component	A class library project for Razor Pages used in multiple projects.
Northwind.Mvc	An ASP.NET Core project for a complex website that uses the MVC pattern and can be more easily unit tested.
Northwind.WebApi	An ASP.NET Core project for an HTTP API service. A good choice for integrating with websites because it can use any JavaScript library or Blazor to interact with the service.
Northwind.BlazorServer	An ASP.NET Core Blazor Server project.
Northwind.BlazorWasm.Client	An ASP.NET Core Blazor WebAssembly client-side project.
Northwind.BlazorWasm.Server	An ASP.NET Core Blazor WebAssembly server-side project.

Building an entity model for use in the rest of the book

Practical applications usually need to work with data in a relational database or another data store. In this chapter, we will define an entity data model for the Northwind database stored in SQL Server or SQLite. It will be used in most of the apps that we create in subsequent chapters.

The `Northwind4SQLServer.sql` and `Northwind4SQLite.sql` script files are different. The script for SQL Server creates 13 tables as well as related views and stored procedures. The script for SQLite is a simplified version that only creates 10 tables because SQLite does not support as many features. The main projects in this book only need those 10 tables so you can complete every task in this book with either database.

Instructions to install SQLite can be found in *Chapter 10, Working with Data Using Entity Framework Core*. In that chapter, you will also find instructions for installing the `dotnet-ef` tool, which you will use to scaffold an entity model from an existing database.

Instructions to install SQL Server can be found in the GitHub repository for the book at the following link: <https://github.com/markjprice/cs11dotnet7/blob/main/docs/sql-server/README.md>.



Good Practice: You should create a separate class library project for your entity data models. This allows easier sharing between backend web servers and frontend desktop, mobile, and Blazor WebAssembly clients.

Creating a class library for entity models using SQLite

You will now define entity data models in a class library so that they can be reused in other types of projects including client-side app models. If you are not using SQL Server, you will need to create this class library for SQLite. If you are using SQL Server, then you can create both a class library for SQLite and one for SQL Server and then switch between them as you choose.

We will automatically generate some entity models using the EF Core command-line tool:

1. Use your preferred code editor to create a new project, as defined in the following list:
 - Project template: **Class Library/classlib**
 - Project file and folder: **Northwind.Common.EntityModels.Sqlite**
 - Workspace/solution file and folder: **PracticalApps**
2. In the **Northwind.Common.EntityModels.Sqlite** project, add package references for the SQLite database provider and EF Core design-time support, as shown in the following markup:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Sqlite" Version="7.0.0" />
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Design" Version="7.0.0">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers;
      buildtransitive</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

3. Delete the **Class1.cs** file.
4. Build the project.
5. Create the **Northwind.db** file for SQLite by copying the **Northwind4SQLite.sql** file into the **PracticalApps** folder (not the project folder!), and then enter the following command at a command prompt or terminal:

```
sqlite3 Northwind.db -init Northwind4SQLite.sql
```

6. Be patient because this command might take a while to create the database structure, as shown in the following output:

```
-- Loading resources from Northwind4SQLite.sql
SQLite version 3.35.5 2022-04-19 14:49:49
Enter ".help" for usage hints.
sqlite>
```

7. To exit SQLite command mode, press *Ctrl + C* on Windows or *Cmd + D* on macOS.
8. Open a command prompt or terminal for the `Northwind.Common.EntityModels.Sqlite` folder.
9. At the command line, generate entity class models for all tables, as shown in the following commands:

```
dotnet ef dbcontext scaffold "Filename=../Northwind.db" Microsoft.  
EntityFrameworkCore.Sqlite --namespace Packt.Shared --data-annotations
```

Note the following:

- The command to perform: `dbcontext scaffold`
- The connection strings. `"Filename=../Northwind.db"`
- The database provider: `Microsoft.EntityFrameworkCore.Sqlite`
- The namespace: `--namespace Packt.Shared`
- To use data annotations as well as the Fluent API: `--data-annotations`

10. Note the build messages and warnings, as shown in the following output:

```
Build started...  
Build succeeded.  
To protect potentially sensitive information in your connection string,  
you should move it out of source code. You can avoid scaffolding the  
connection string by using the Name= syntax to read it from configuration  
- see https://go.microsoft.com/fwlink/?LinkId=2131148. For more  
guidance on storing connection strings, see http://go.microsoft.com/fwlink/?LinkId=723263.
```

Improving the class-to-table mapping

The `dotnet-ef` command-line tool generates different code for SQL Server and SQLite because they support different levels of functionality.

For example, SQL Server text columns can have limits on the number of characters. SQLite does not support this. So, `dotnet-ef` will generate validation attributes to ensure `string` properties are limited to a specified number of characters for SQL Server but not for SQLite, as shown in the following code:

```
// SQLite database provider-generated code  
[Column(TypeName = "nvarchar (15)")]  
public string CategoryName { get; set; } = null!;  
  
// SQL Server database provider-generated code  
[StringLength(15)]  
public string CategoryName { get; set; } = null!;
```

Neither database provider will mark non-nullable `string` properties as required:

```
// no runtime validation of non-nullable property  
public string CategoryName { get; set; } = null!;  
  
// nullable property
```

```
public string? Description { get; set; }

// decorate with attribute to perform runtime validation
[Required]
public string CategoryName { get; set; } = null!;
```

We will make some small changes to improve the entity model mapping and validation rules for SQLite.



Remember that all code is available in the GitHub repository for the book. Although you will learn more by typing code yourself, you never have to. Go to the following link and press . to get a live code editor in your browser: <https://github.com/markjprice/cs11dotnet7>.

First, we will add a regular expression to validate that a `CustomerId` value is exactly five uppercase letters. Second, we will add string length requirements to validate that multiple properties throughout the entity models know the maximum length allowed for their text values:

1. In `Customer.cs`, add a regular expression to validate its primary key value to only allow uppercase Western characters, as shown highlighted in the following code:

```
[Key]
[Column(TypeName = "nchar (5)")]
[RegularExpression("[A-Z]{5}")]
public string CustomerId { get; set; } = null!;
```

2. Activate your code editor's find and replace feature (in Visual Studio 2022, navigate to **Edit | Find and Replace | Quick Replace**), toggle on **Use Regular Expressions**, and then type a regular expression in the find box, as shown in *Figure 12.2* and in the following expression:

```
\[Column\](TypeName = "(nchar|nvarchar) \((.*\)\)"\)\]
```

3. In the replace box, type a replacement regular expression, as shown in the following expression:

```
$&\n    [StringLength($2)]
```

After the newline character, `\n`, I have included four space characters to indent correctly on my system, which uses two space characters per indentation level. You can insert as many as you wish.

4. Set the find and replace to search files in the current project.
5. Execute the search and replace to replace all, as shown in *Figure 12.2*:

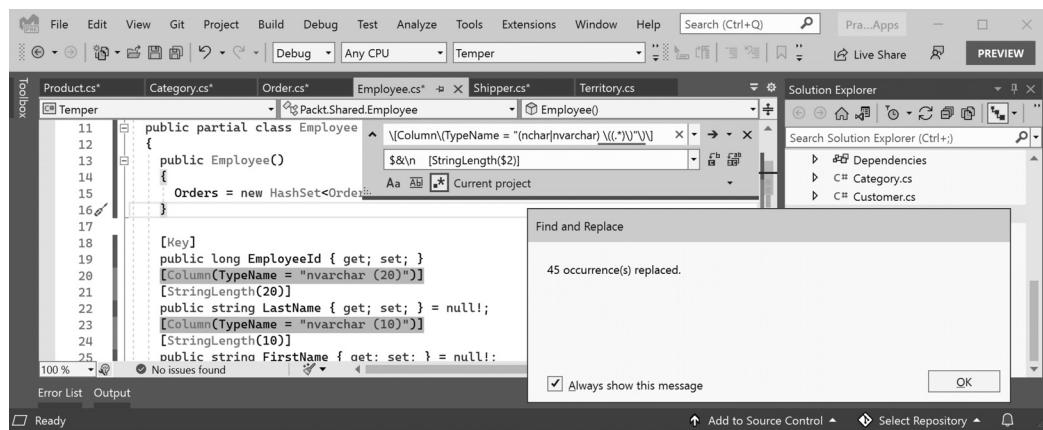


Figure 12.2: Search and replace all matches using regular expressions in Visual Studio 2022

6. Change any date/time properties, for example, in `Employee.cs`, to use a nullable `DateTime` instead of an array of bytes, as shown in the following code:

```
// before
[Column(TypeName = "datetime")]
public byte[]? BirthDate { get; set; }

// after
[Column(TypeName = "datetime")]
public DateTime? BirthDate { get; set; }
```



Use your code editor's find feature to search for "datetime" to find all the properties that need changing.

7. Change any `money` properties, for example, in `Order.cs`, to use a nullable `decimal` instead of an array of bytes, as shown in the following code:

```
// before
[Column(TypeName = "money")]
public byte[]? Freight { get; set; }

// after
[Column(TypeName = "money")]
public decimal? Freight { get; set; }
```



Use your code editor's find feature to search for "money" to find all the properties that need changing.

8. In `Product.cs`, make the `Discontinued` property a `bool` instead of an array of bytes and remove the initializer that sets the default value to `null`, as shown in the following code:

```
[Column(TypeName = "bit")]
public bool Discontinued { get; set; }
```

9. In `Category.cs`, make the `CategoryId` property an `int`, as shown highlighted in the following code:

```
[Key]
public int CategoryId { get; set; }
```

10. In `Category.cs`, make the `CategoryName` property required, as shown highlighted in the following code:

```
[Required]
[Column(TypeName = "nvarchar (15)")]
[StringLength(15)]
public string CategoryName { get; set; }
```

11. In `Customer.cs`, make the `CompanyName` property required, as shown highlighted in the following code:

```
[Required]
[Column(TypeName = "nvarchar (40)")]
[StringLength(40)]
public string CompanyName { get; set; }
```

12. In `Employee.cs`, make the:

- `EmployeeId` property an `int` instead of a `long`.
- `FirstName` and `LastName` properties required.
- `ReportsTo` property an `int?` instead of a `long?`.

13. In `EmployeeTerritory.cs`, make the:

- `EmployeeId` property an `int` instead of a `long`.
- `TerritoryId` property required.

14. In `Order.cs`:

- Make the `OrderId` property an `int` instead of a `long`.
- Decorate the `CustomerId` property with a regular expression to enforce five uppercase characters.

- Make the `EmployeeId` property an `int?` instead of a `long?`.
- Make the `ShipVia` property an `int?` instead of a `long?`.

15. In `OrderDetail.cs`, make the:

- `OrderId` property an `int` instead of a `long`.
- `ProductId` property an `int` instead of a `long`.
- `Quantity` property a `short` instead of a `long`.

16. In `Product.cs`, make the:

- `ProductId` property an `int` instead of a `long`.
- `ProductName` property required.
- `SupplierId` and `CategoryId` properties an `int?` instead of a `long?`.
- `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` properties a `short?` instead of a `long?`.

17. In `Shipper.cs`, make the:

- `ShipperId` property an `int` instead of a `long`.
- `CompanyName` property required.

18. In `Supplier.cs`, make the:

- `SupplierId` property an `int` instead of a `long`.
- `CompanyName` property required.

19. In `Territory.cs`, make the:

- `RegionId` property an `int` instead of a `long`.
- `TerritoryId` and `TerritoryDescription` properties required.

Now that we have a class library for the entity classes, we can create a class library for the database context.

Creating a class library for a Northwind database context

You will now define a database context class library:

1. Add a new project to the solution/workspace, as defined in the following list:
 - Project template: **Class Library**/`classlib`
 - Project file and folder: `Northwind.Common.DataContext.Sqlite`
 - Workspace/solution file and folder: `PracticalApps`
2. In Visual Studio, set the startup project for the solution to the current selection. In Visual Studio Code, select `Northwind.Common.DataContext.Sqlite` as the active OmniSharp project.

3. In the `Northwind.Common.DataContext.Sqlite` project, add a project reference to the `Northwind.Common.EntityModels.Sqlite` project and add a package reference to the EF Core data provider for SQLite, as shown in the following markup:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SQLite" Version="7.0.0" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include=
    "...\\Northwind.Common.EntityModels.Sqlite\\Northwind.Common
    .EntityModels.Sqlite.csproj" />
</ItemGroup>
```



Warning! The path to the project reference should not have a line break in your project file.

4. In the `Northwind.Common.DataContext.Sqlite` project, delete the `Class1.cs` class file.
5. Build the `Northwind.Common.DataContext.Sqlite` project.
6. Move the `NorthwindContext.cs` file from the `Northwind.Common.EntityModels.Sqlite` project/folder to the `Northwind.Common.DataContext.Sqlite` project/folder.



In Visual Studio **Solution Explorer**, if you drag and drop a file between projects it will be copied. If you hold down *Shift* while dragging and dropping, it will be moved. In Visual Studio Code **EXPLORER**, if you drag and drop a file between projects it will be moved. If you hold down *Ctrl* while dragging and dropping, it will be copied.

7. In `NorthwindContext.cs`, in the `OnConfiguring` method, remove the compiler `#warning` about the connection string.



Good Practice: We will override the default database connection string in any projects such as websites that need to work with the Northwind database, so the class derived from `DbContext` must have a constructor with a `DbContextOptions` parameter for this to work, and the generated file does this correctly, as shown in the following code:

```
public NorthwindContext(DbContextOptions<NorthwindContext>
  options)
  : base(options)
{}
```

8. In `NorthwindContext.cs`, in the `OnConfiguring` method, add statements to check the end of the current directory to adjust for when running in Visual Studio 2022 compared to the command line and Visual Studio Code, as shown highlighted in the following code:

```
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        string dir = Environment.CurrentDirectory;
        string path = string.Empty;

        if (dir.EndsWith("net7.0"))
        {
            // Running in the <project>\bin\<Debug/Release>\net7.0 directory.
            path = Path.Combine("../", "..", "..", "..", "Northwind.db");
        }
        else
        {
            // Running in the <project> directory.
            path = Path.Combine("../", "Northwind.db");
        }

        optionsBuilder.UseSqlite($"Filename={path}");
    }
}
```

9. In the `OnModelCreating` method, remove all Fluent API statements that call the `ValueGeneratedNever` method, like the one shown in the following code. This will configure primary key properties like `SupplierId` to never generate a value automatically or call the `HasDefaultValueSql` method:

```
modelBuilder.Entity<Supplier>(entity =>
{
    entity.Property(e => e.SupplierId).ValueGeneratedNever();
});
```



If we do not remove the configuration like the statements above, then when we add new suppliers, the `SupplierId` value will always be 0 and we will only be able to add one supplier with that value; all other attempts will throw an exception.

10. For the `Product` entity, tell SQLite that the `UnitPrice` can be converted from `decimal` to `double`. The `OnModelCreating` method should now be much simplified, as shown in the following code:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
```

```
modelBuilder.Entity<OrderDetail>(entity =>
{
    entity.HasKey(e => new { e.OrderId, e.ProductId });

    entity.HasOne(d => d.Order)
        .WithMany(p => p.OrderDetails)
        .HasForeignKey(d => d.OrderId)
        .OnDelete(DeleteBehavior.ClientSetNull);

    entity.HasOne(d => d.Product)
        .WithMany(p => p.OrderDetails)
        .HasForeignKey(d => d.ProductId)
        .OnDelete(DeleteBehavior.ClientSetNull);
});

modelBuilder.Entity<Product>()
    .Property(product => product.UnitPrice)
    .HasConversion<double>();

OnModelCreatingPartial(modelBuilder);
}
```

11. In the `Northwind.Common.DataContext.Sqlite` project, add a class named `NorthwindContextExtensions.cs`. Modify its contents to define an extension method that adds the Northwind database context to a collection of dependency services, as shown in the following code:

```
using Microsoft.EntityFrameworkCore; // UseSqlite
using Microsoft.Extensions.DependencyInjection; // IServiceCollection

namespace Packt.Shared;

public static class NorthwindContextExtensions
{
    /// <summary>
    /// Adds NorthwindContext to the specified IServiceCollection. Uses the
    /// Sqlite database provider.
    /// </summary>
    /// <param name="services"></param>
    /// <param name="relativePath">Set to override the default of ".."
    /// </param>
    /// <returns>An IServiceCollection that can be used to add more services.
    /// </returns>
    public static IServiceCollection AddNorthwindContext(
        this IServiceCollection services, string relativePath = "..")
    {
        string databasePath = Path.Combine(relativePath, "Northwind.db");
    }
}
```

```
services.AddDbContext<NorthwindContext>(options =>
{
    options.UseSqlite($"Data Source={databasePath}");

    options.LogTo(WriteLine, // Console
        new[] { Microsoft.EntityFrameworkCore
            .Diagnostics.RelationalEventId.CommandExecuting });
});

return services;
}
}
```

12. Build the two class libraries and fix any compiler errors.

Creating a class library for entity models using SQL Server

To use SQL Server, you will not need to do anything if you already set up the Northwind database in *Chapter 10, Working with Data Using Entity Framework Core*. But you will now create the entity models using the `dotnet-e` tool:

1. Add a new project, as defined in the following list:
 - Project template: **Class Library/classlib**
 - Project file and folder: `Northwind.Common.EntityModels.SqlServer`
 - Workspace/solution file and folder: `PracticalApps`
2. In the `Northwind.Common.EntityModels.SqlServer` project, add package references for the SQL Server database provider and EF Core design-time support, as shown in the following markup:

```
<ItemGroup>
    <PackageReference
        Include="Microsoft.EntityFrameworkCore.SqlServer" Version="7.0.0" />
    <PackageReference
        Include="Microsoft.EntityFrameworkCore.Design" Version="7.0.0">
        <PrivateAssets>all</PrivateAssets>
        <IncludeAssets>runtime; build; native; contentfiles; analyzers;
        buildtransitive</IncludeAssets>
    </PackageReference>
</ItemGroup>
```

3. Delete the `Class1.cs` file.
4. Build the project.
5. Open a command prompt or terminal for the `Northwind.Common.EntityModels.SqlServer` folder.

6. At the command line, generate entity class models for all tables, as shown in the following commands:

```
dotnet ef dbcontext scaffold "Data Source=.;Initial Catalog=Northwind;Integrated Security=true;" Microsoft.EntityFrameworkCore.SqlServer --namespace Packt.Shared --data-annotations
```

Note the following:

- The command to perform: `dbcontext scaffold`
- The connection strings. `"Data Source=.;Initial Catalog=Northwind;Integrated Security=true;"`
- The database provider: `Microsoft.EntityFrameworkCore.SqlServer`
- The namespace: `--namespace Packt.Shared`
- To use data annotations as well as the Fluent API: `--data-annotations`

7. In `Customer.cs`, add a regular expression to validate its primary key value to only allow uppercase Western characters, as shown highlighted in the following code:

```
[Key]  
[StringLength(5)]  
[RegularExpression("[A-Z]{5}")]  
public string CustomerId { get; set; } = null!;
```

8. In `Customer.cs`, make the `CustomerId` and `CompanyName` properties required.
9. Add a new project, as defined in the following list:
 - Project template: **Class Library/classlib**
 - Project file and folder: `Northwind.Common.DataContext.SqlServer`
 - Workspace/solution file and folder: `PracticalApps`
 - In Visual Studio Code, select `Northwind.Common.DataContext.SqlServer` as the active OmniSharp project
10. In the `Northwind.Common.DataContext.SqlServer` project, add a project reference to the `Northwind.Common.EntityModels.SqlServer` project and add a package reference to the EF Core data provider for SQL Server, as shown in the following markup:

```
<ItemGroup>  
  <PackageReference  
    Include="Microsoft.EntityFrameworkCore.SqlServer" Version="7.0.0" />  
  </ItemGroup>  
  
<ItemGroup>  
  <ProjectReference Include=  
    "..\Northwind.Common.EntityModels.SqlServer\Northwind.Common  
.EntityModels.SqlServer.csproj" />  
  </ItemGroup>
```



Warning! The path to the project reference should not have a line break in your project file.

11. In the `Northwind.Common.DataContext.SqlServer` project, delete the `Class1.cs` file.
12. Build the `Northwind.Common.DataContext.SqlServer` project.
13. Move the `NorthwindContext.cs` file from the `Northwind.Common.EntityModels.SqlServer` project/folder to the `Northwind.Common.DataContext.SqlServer` project/folder.
14. In the `Northwind.Common.DataContext.SqlServer` project, in `NorthwindContext.cs`, remove the compiler warning about the connection string.
15. In the `Northwind.Common.DataContext.SqlServer` project, add a class named `NorthwindContextExtensions.cs`. Modify its contents to define an extension method that adds the Northwind database context to a collection of dependency services, as shown in the following code:

```
using Microsoft.EntityFrameworkCore; // UseSqlServer
using Microsoft.Extensions.DependencyInjection; // IServiceCollection

namespace Packt.Shared;

public static class NorthwindContextExtensions
{
    /// <summary>
    /// Adds NorthwindContext to the specified IServiceCollection. Uses the
    /// SqlServer database provider.
    /// </summary>
    /// <param name="services"></param>
    /// <param name="connectionString">Set to override the default.</param>
    /// <returns>An IServiceCollection that can be used to add more
    /// services.</returns>
    public static IServiceCollection AddNorthwindContext(
        this IServiceCollection services,
        string connectionString = "Data Source=.;Initial Catalog=Northwind;" +
        "Integrated Security=true;MultipleActiveResultsets=true;Encrypt=false")
    {
        services.AddDbContext<NorthwindContext>(options =>
        {
            options.UseSqlServer(connectionString);

            options.LogTo(WriteLine, // Console
                new[] { Microsoft.EntityFrameworkCore
                    .Diagnostics.RelationalEventId.CommandExecuting });
        });
    }
}
```

```
    });

    return services;
}

}
```

16. Build the two class libraries and fix any compiler errors.



Good Practice: We have provided optional arguments for the `AddNorthwindContext` method so that we can override the hardcoded SQLite database filename path or the SQL Server database connection string. This will allow us more flexibility, for example, to load these values from a configuration file.

Testing the class libraries

Now let's build some unit tests to ensure the class libraries are working correctly:

1. Use your preferred coding tool to add a new **xUnit Test Project [C#]/xunit** project named `Northwind.Common.UnitTests` to the `PracticalApps` workspace/solution.
2. In the `Northwind.Common.UnitTests` project, add a project reference to the `Northwind.Common.DataContext` project for either SQLite or SQL Server, as shown highlighted in the following configuration:

```
<ItemGroup>
  <!-- change Sqlite to SqlServer if you prefer -->
  <ProjectReference Include=".\\Northwind.Common.DataContext.Sqlite\\
  Northwind.Common.DataContext.Sqlite.csproj" />
</ItemGroup>
```



Warning! The project reference must go all on one line with no line break.

3. Build the `Northwind.Common.UnitTests` project.
4. Rename `UnitTest1.cs` to `EntityModelTests.cs`.
5. Modify the contents of the file to define two tests, the first to connect to the database and the second to confirm there are eight categories in the database, as shown in the following code:

```
using Packt.Shared; // NorthwindContext

namespace Northwind.Common.UnitTests
{
    public class EntityModelTests
    {
```

```
[Fact]
public void DatabaseConnectTest()
{
    using (NorthwindContext db = new())
    {
        Assert.True(db.Database.CanConnect());
    }
}

[Fact]
public void CategoryCountTest()
{
    using (NorthwindContext db = new())
    {
        int expected = 8;
        int actual = db.Categories.Count();

        Assert.Equal(expected, actual);
    }
}
```

6. Run the unit tests:

- If you are using Visual Studio 2022, then navigate to **Test | Run All Tests** in **Test Explorer**.
- If you are using Visual Studio Code, then in the `Northwind.Common.UnitTests` project's **TERMINAL** window, run the tests, as shown in the following command: `dotnet test`.

7. Note that the results should indicate that two tests ran, and both passed. If either of the two tests fail, then fix the issue. For example, if you are using SQLite then check the `Northwind.db` file is in the solution directory (one up from the project directories.)

Understanding web development

Developing for the web means developing with **Hypertext Transfer Protocol (HTTP)**, so we will start by reviewing this important foundational technology.

Understanding Hypertext Transfer Protocol

To communicate with a web server, the client, also known as the **user agent**, makes calls over the network using HTTP. As such, HTTP is the technical underpinning of the web. So, when we talk about websites and web services, we mean that they use HTTP to communicate between a client (often a web browser) and a server.

A client makes an HTTP request for a resource, such as a page, uniquely identified by a **Uniform Resource Locator (URL)**, and the server sends back an HTTP response, as shown in *Figure 12.3*:

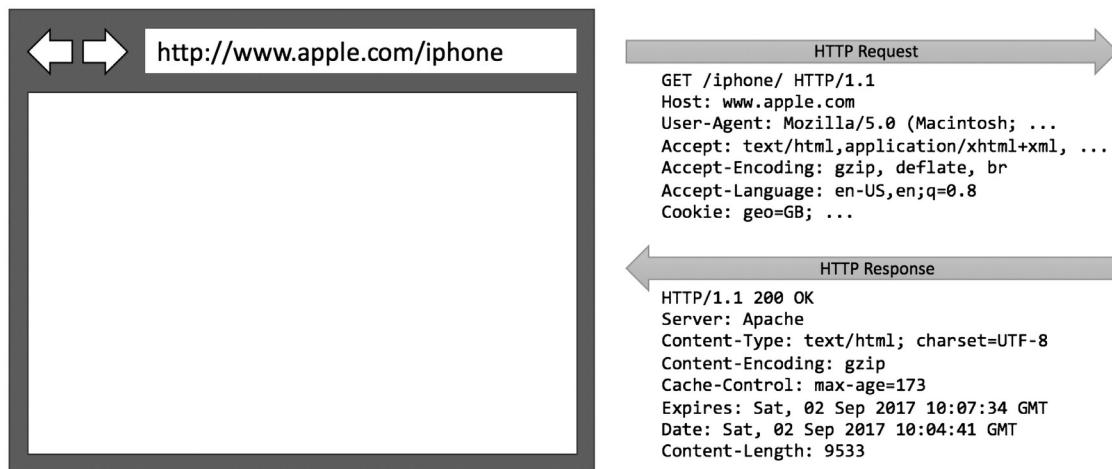


Figure 12.3: An HTTP request and response

You can use Google Chrome and other browsers to record requests and responses.



Good Practice: Google Chrome is currently used by about two thirds of website visitors worldwide, and it has powerful, built-in developer tools, so it is a good first choice for testing your websites. Test your websites with Chrome and at least two other browsers, for example, Firefox and Safari for macOS and iPhone. Microsoft Edge switched from using Microsoft's own rendering engine to using Chromium in 2019, so it is less important to test with it. If Microsoft's Internet Explorer is used at all, it tends to mostly be inside organizations for intranets.

Understanding the components of a URL

A Uniform Resource Locator (URL) is made up of several components:

- **Scheme:** `http` (clear text) or `https` (encrypted).
- **Domain:** For a production website or service, the **top-level domain (TLD)** might be `example.com`. You might have subdomains such as `www`, `jobs`, or `extranet`. During development, you typically use `localhost` for all websites and services.
- **Port number:** For a production website or service, `80` for `http`, `443` for `https`. These port numbers are usually inferred from the scheme. During development, other port numbers are commonly used, such as `5000`, `5001`, and so on, to differentiate between websites and services that all use the shared domain `localhost`.
- **Path:** A relative path to a resource, for example, `/customers/germany`.
- **Query string:** A way to pass parameter values, for example, `?country=Germany&searchtext=shoes`.
- **Fragment:** A reference to an element on a web page using its `id`, for example, `#toc`.



URL is a subset of **Uniform Resource Identifier (URI)**. A URL specifies where a resource is located and how to get it. A URI identifies a resource either by URL or URN (**Uniform Resource Name**).

Assigning port numbers for projects in this book

In this book, we will use the domain `localhost` for all websites and web services, so we will use port numbers to differentiate projects when multiple need to execute at the same time, as shown in the following table:

Project	Description	Port numbers
<code>Northwind.Web</code>	ASP.NET Core Razor Pages website	5000 HTTP, 5001 HTTPS
<code>Northwind.Mvc</code>	ASP.NET Core MVC website	5000 HTTP, 5001 HTTPS
<code>Northwind.WebApi</code>	ASP.NET Core Web API service	5002 HTTPS
<code>Minimal.WebApi</code>	ASP.NET Core Web API (minimal)	5003 HTTPS
<code>Northwind.BlazorServer</code>	ASP.NET Core Blazor Server	5004 HTTP, 5005 HTTPS
<code>Northwind.BlazorWasm</code>	ASP.NET Core Blazor WebAssembly	5006 HTTP, 5007 HTTPS

Using Google Chrome to make HTTP requests

Let's explore how to use Google Chrome to make HTTP requests:

1. Start Google Chrome.
2. Navigate to **More tools | Developer tools**.
3. Click the **Network** tab, and Chrome should immediately start recording the network traffic between your browser and any web servers (note the red circle), as shown in *Figure 12.4*:

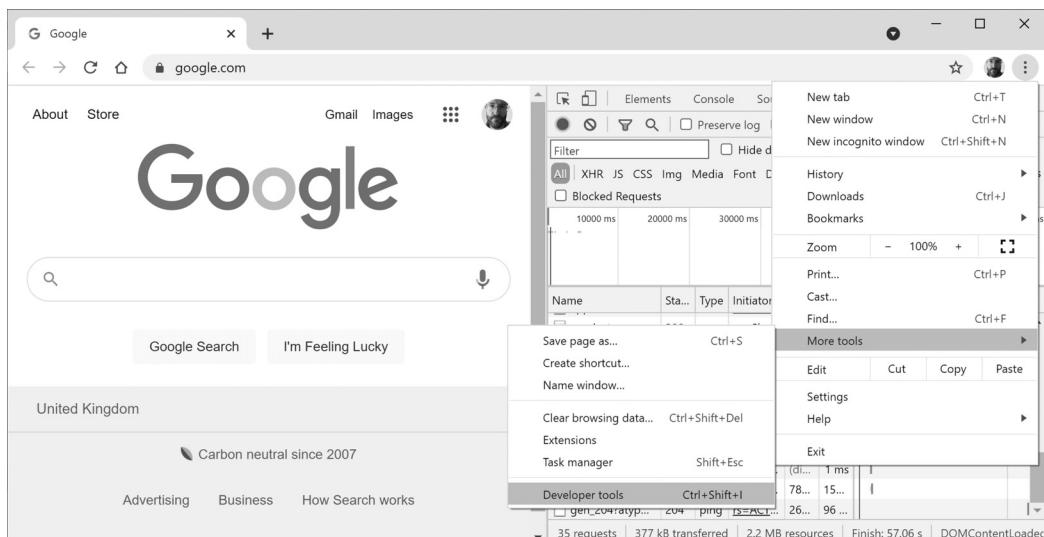


Figure 12.4: Chrome Developer Tools recording network traffic

4. In Chrome's address box, enter the address of Microsoft's website for learning ASP.NET, as shown in the following URL:
<https://dotnet.microsoft.com/learn/aspnet>
5. In **Developer Tools**, in the list of recorded requests, scroll to the top and click on the first entry, the row where the **Type** is **document**, as shown in *Figure 12.5*:

Figure 12.5: Recorded requests in Developer Tools

6. On the right-hand side, click on the **Headers** tab, and you will see details about **Request Headers** and **Response Headers**, as shown in *Figure 12.6*:

Figure 12.6: Request and response headers

Note the following aspects:

- **Request Method** is `GET`. Other HTTP methods that you could see here include `POST`, `PUT`, `DELETE`, `HEAD`, and `PATCH`.
- **Status Code** is `200 OK`. This means that the server found the resource that the browser requested and has returned it in the body of the response. Other status codes that you might see in response to a `GET` request include `301 Moved Permanently`, `400 Bad Request`, `401 Unauthorized`, and `404 Not Found`.
- **Request Headers** sent by the browser to the web server include:
 - `accept`, which lists what formats the browser accepts. In this case, the browser is saying it understands `HTML`, `XHTML`, `XML`, and some image formats, but it will accept all other files `(*/*)`. Default weightings, also known as quality values, are `1.0`. `XML` is specified with a quality value of `0.9` so it is preferred less than `HTML` or `XHTML`. All other file types are given a quality value of `0.8` so are least preferred.
 - `accept-encoding`, which lists what compression algorithms the browser understands, in this case, `GZIP`, `DEFLATE`, and `Brotli`.
 - `accept-language`, which lists the human languages it would prefer the content to use. In this case, `US English`, which has a default quality value of `1.0`, then any dialect of `English` that has an explicitly specified quality value of `0.9`, and then any dialect of `Swedish` that has an explicitly specified quality value of `0.8`.
- **Response Headers**, `content-encoding`, which tells me the server has sent back the `HTML` web page response compressed using the `GZIP` algorithm because it knows that the client can decompress that format. (This is not visible in *Figure 12.6* because there is not enough space to expand the **Response Headers** section.)

7. Close Chrome.

Understanding client-side web development technologies

When building websites, a developer needs to know more than just `C#` and `.NET`. On the client (that is, in the web browser), you will use a combination of the following technologies:

- **HTML5**: This is used for the content and structure of a web page.
- **CSS3**: This is used for the styles applied to elements on the web page.
- **JavaScript**: This is used to code any business logic needed on the web page, for example, validating form input or making calls to a web service to fetch more data needed by the web page.

Although `HTML5`, `CSS3`, and `JavaScript` are the fundamental components of frontend web development, there are many additional technologies that can make frontend web development more productive, including:

- **Bootstrap**, the world's most popular frontend open-source toolkit.
- **SASS** and **LESS**, CSS preprocessors for styling.
- Microsoft's **TypeScript** language for writing more robust code.
- JavaScript libraries such as **Angular**, **jQuery**, **React**, and **Vue**.

All these higher-level technologies ultimately translate or compile to the underlying three core technologies, so they work across all modern browsers.

As part of the build and deploy process, you will likely use technologies such as:

- **Node.js**, a framework for server-side development using JavaScript.
- **Node Package Manager (npm)** and **Yarn**, both client-side package managers.
- **Webpack**, a popular module bundler, a tool for compiling, transforming, and bundling website source files.

Practicing and exploring

Test your knowledge and understanding by answering some questions and exploring this chapter's topics with deeper research.

Exercise 12.1 – Test your knowledge

Answer the following questions:

1. What was the name of Microsoft first dynamic server-side executed web page technology and why is it still useful to know this history today?
2. What are the names of two Microsoft web servers?
3. What are some differences between a microservice and a nanoservice?
4. What is Blazor?
5. What was the first version of ASP.NET Core that could not be hosted on .NET Framework?
6. What is a user agent?
7. What impact does the HTTP request-response communication model have on web developers?
8. Name and describe four components of a URL.
9. What capabilities does Developer Tools give you?
10. What are the three main client-side web development technologies and what do they do?

Exercise 12.2 – Know your webabbreviations

What do the following web abbreviations stand for and what do they do?

1. URI
2. URL
3. WCF
4. TLD
5. API
6. SPA
7. CMS
8. Wasm

- 9. SASS
- 10. REST

Exercise 12.3 – Explore topics

Use the links on the following page to learn more detail about the topics covered in this chapter:

<https://github.com/markjprice/cs11dotnet7/blob/main/book-links.md#chapter-12---introducing-web-development-using-aspnet-core>

Summary

In this chapter, you have:

- Been introduced to some of the app models and workloads that you can use to build websites and web services using C# and .NET.
- Created two to four class libraries to define an entity data model for working with the Northwind database using either SQLite or SQL Server or both.

In the following chapters, you will learn the details about how to build the following:

- Simple websites using static HTML pages and dynamic Razor Pages.
- Complex websites using the Model-View-Controller (MVC) design pattern.
- Web services that can be called by any platform that can make an HTTP request, and client websites that call those web services.
- Blazor user interface components that can be hosted on a web server, in the browser, or on hybrid web-native mobile and desktop apps.