MANNING

ASP.NET Core IN ACTION

SECOND EDITION

MEAP

Andrew Lock

MEAP Edition
Manning Early Access Program

ASP.NET Core in Action

Second Edition

Version 5

For more information on this and other Manning titles go to
manning.com

welcome

Thanks for purchasing the MEAP for *ASP.NET Core in Action, Second Edition*. This book has been written to take a moderately experienced C# developer, without web development experience, to being a well-round ASP.NET Core developer, ready to build your first web applications.

The genesis of ASP.NET Core goes back several years, but I first really started paying attention shortly before the release of ASP.NET Core RC2. As an ASP.NET developer for many years, the new direction and approach to development Microsoft were embracing was a breath of fresh air. No longer are Visual Studio and Windows mandated. You are now free to build and run cross-platform .NET applications on any OS, with any IDE, and using the tools that you choose.

Along with that, the entire framework is open-source, and developed with many contributions from the community. It was this aspect that drew me in initially. The ability to wade through the framework code and see how features were implemented was such a revelation (compared to the dubious documentation of old) that I was hooked!

ASP.NET Core has grown massively since its release in 2016—.NET Core 3, released in 2019, was the fastest adopted version of .NET ever. *ASP.NET Core in Action* is my attempt to get you started and productive with the latest version of the framework as soon as possible. In the first half of the book, you will work through the basics of a typical ASP.NET Core application, focusing on how to create basic web pages using Razor Pages and Web APIs using MVC controllers.

In the second half, you will build on this core knowledge, looking at more advanced requirements and how to add extra features to your application. You will learn how to secure your application behind a login screen, how to handle configuration and dependency injection, and how to deploy your application to production. In the last part of the book you will look in depth at further bending the framework to your will by creating custom components.

There's already a lot of ground to cover in such a big framework, so I won't be covering Blazor in this book, but I will be updating the book to cover .NET 5 before final publication. Your feedback is essential to creating the best book possible, so please be sure to post any comments, questions or suggestions you have about the book in the liveBook discussion forum. I appreciate knowing where to make improvements to ensure you can get the most out of it!

Dr Andrew Lock

brief contents

PART 1: BUILDING YOUR FIRST APPLICATIONS

PART 2: BUILDING COMPLETE APPLICATIONS

PART 3: EXTENDING YOUR APPLICATION

APPENDIXES

Part 1

Getting started with ASP.NET Core

Web applications are everywhere these days, from social media web apps and news sites, to the apps on your phone. Behind the scenes, there is almost always a server running a web application or an HTTP API. Web applications are expected to be infinitely scalable, deployed to the cloud, and highly performant. Getting started can be overwhelming at the best of times and doing so with such high expectations can be even more of a challenge.

The good news for you as readers is that ASP.NET Core was designed to meet those requirements. Whether you need a simple website, a complex e-commerce web app, or a distributed web of microservices, you can use your knowledge of ASP.NET Core to build lean web apps that fit your needs. ASP.NET Core lets you build and run web apps on Windows, Linux, or macOS. It's highly modular, so you only use the components you need, keeping your app as compact and performant as possible.

In part 1, you'll go from a standing start all the way to building your first web applications and APIs. Chapter 1 gives a high-level overview of ASP.NET Core, which you'll find especially useful if you're new to web development in general. You'll get your first glimpse of a full ASP.NET Core application in chapter 2, in which we look at each component of the app in turn and see how they work together to generate a response.

Chapter 3 looks in detail at the middleware pipeline, which defines how incoming web requests are processed and how a response is generated. We look at several standard pieces of middleware and see how the Razor Pages framework fits in to the pipeline. In Chapters 4 through 8 we focus on Razor Pages, which is the main approach to generate responses in ASP.NET Core apps. In chapters 4 through 6 we examine the behavior of the Razor Pages framework itself, routing, and model binding. In Chapters 7 and 8, we look at how to build the UI for your application using the Razor syntax and Tag Helpers, so that users can navigate and interact with your app. Finally, in chapter 9, we explore specific features of ASP.NET Core that let you build Web APIs, and how that differs from building UI-based applications.

There’s a lot of content in part 1, but by the end, you’ll be well on your way to building simple applications with ASP.NET Core. Inevitably, I gloss over some of the more complex configuration aspects of the framework, but you should get a good understanding of the Razor Pages framework and how you can use it to build dynamic web apps. In later parts of this book, we’ll dive deeper into the ASP.NET Core framework, where you’ll learn how to configure your application and add extra features, such as user profiles.

1

Getting started with ASP.NET Core

This chapter covers

- **What is ASP.NET Core?**
- **Things you can build with ASP.NET Core**
- **The advantages and limitations of .NET Core**
- **How ASP.NET Core works**

Choosing to learn and develop with a new framework is a big investment, so it's important to establish early on whether it's right for you. In this chapter, I provide some background about ASP.NET Core, what it is, how it works, and why you should consider it for building your web applications.

If you're new to .NET development, this chapter will help you to understand the .NET landscape. For existing .NET developers, I provide guidance on whether now is the right time to consider moving your focus to .NET Core, and the advantages ASP.NET Core can bring over previous versions of ASP.NET.

By the end of this chapter, you should have a good overview of the .NET landscape, the role of .NET Core, and the basic mechanics of how ASP.NET Core works—so without further ado, let's dive in!

1.1 An introduction to ASP.NET Core

ASP.NET Core is the latest evolution of Microsoft's popular ASP.NET web framework, released in June 2016. Previous versions of ASP.NET have seen many incremental updates, focusing on high developer productivity and prioritizing backwards compatibility. ASP.NET Core bucks that trend by making significant architectural changes that rethink the way the web framework is designed and built.

ASP.NET Core owes a lot to its ASP.NET heritage and many features have been carried forward from before, but ASP.NET Core is a new framework. The whole technology stack has been rewritten, including both the web framework and the underlying platform.

At the heart of the changes is the philosophy that ASP.NET should be able to hold its head high when measured against other modern frameworks, but that existing .NET developers should continue to be left with a sense of familiarity. In this section I cover

- The reasons for using a web framework
- The previous ASP.NET framework's benefits and limitations
- What ASP.NET Core is and its motivations

At the end of this section you should have a good sense of why ASP.NET Core was created, its design goals, and why you might want to use it

1.1.1 Using a web framework

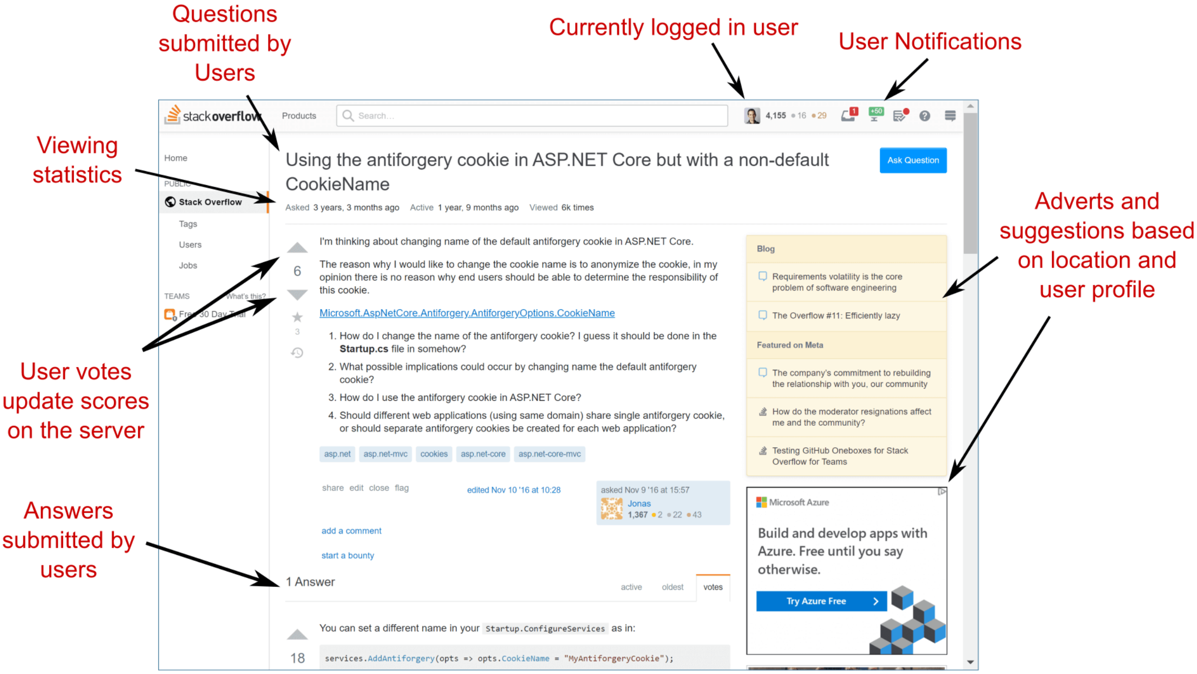
If you're new to web development, it can be daunting moving into an area with so many buzzwords and a plethora of ever-changing products. You may be wondering if they're all necessary—how hard can it be to return a file from a server?

Well, it's perfectly possible to build a static web application without the use of a web framework, but its capabilities will be limited. As soon as you want to provide any kind of security or dynamism, you'll likely run into difficulties, and the original simplicity that enticed you will fade before your eyes!

Just as you may have used desktop or mobile development frameworks for building native applications, ASP.NET Core makes writing web applications faster, easier, and more secure than trying to build everything from scratch. It contains libraries for common things like

- Creating dynamically changing web pages
- Letting users log in to your web app
- Letting users use their Facebook account to log in to your web app using OAuth
- Providing a common structure to build maintainable applications
- Reading configuration files
- Serving image files
- Logging requests made to your web app

The key to any modern web application is the ability to generate dynamic web pages. A *dynamic web page* displays different data depending on the current logged-in user, for example, or it could display content submitted by users. Without a dynamic framework, it wouldn’t be possible to log in to websites or to have any sort of personalized data displayed on a page. In short, websites like Amazon, eBay, and Stack Overflow (seen in figure 1.1) wouldn’t be possible.



Using the antiforgery cookie in ASP.NET Core but with a non-default CookieName

Asked 3 years, 3 months ago Active 1 year, 9 months ago Viewed 6k times

I'm thinking about changing name of the default antiforgery cookie in ASP.NET Core.

The reason why I would like to change the cookie name is to anonymize the cookie, in my opinion there is no reason why end users should be able to determine the responsibility of this cookie.

Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.CookieName

1. How do I change the name of the antiforgery cookie? I guess it should be done in the **Startup.cs** file in somehow?
2. What possible implications could occur by changing name the default antiforgery cookie?
3. How do I use the antiforgery cookie in ASP.NET Core?
4. Should different web applications (using same domain) share single antiforgery cookie, or should separate antiforgery cookies be created for each web application?

edited Nov 10 '16 at 10:28

asked Nov 9 '16 at 15:57
Jonas
1,367 ●2 ●22 ●43

1 Answer

You can set a different name in your `Startup.ConfigureServices` as in:

```
services.AddAntiforgery(opts => opts.CookieName = "MyAntiforgeryCookie");
```

Requirements volatility is the core problem of software engineering

The Overflow #11: Efficiently lazy

The company’s commitment to rebuilding the relationship with you, our community

How do the moderator resignations affect me and the community?

Testing GitHub Oneboxes for Stack Overflow for Teams

Build and develop apps with Azure. Free until you say otherwise.

The Stack Overflow website (https://stackoverflow.com) is built using ASP.NET and is almost entirely dynamic content.

1.1.2 The benefits and limitations of ASP.NET

To understand *why* Microsoft decided to build a new framework, it’s important to understand the benefits and limitations of the previous ASP.NET web framework.

The first version of ASP.NET was released in 2002 as part of .NET Framework 1.0, in response to the then conventional scripting environments of classic ASP and PHP. ASP.NET Web Forms allowed developers to rapidly create web applications using a graphical designer and a simple event model that mirrored desktop application-building techniques.

The ASP.NET framework allowed developers to quickly create new applications, but over time, the web development ecosystem changed. It became apparent that ASP.NET Web Forms suffered from many issues, especially when building larger applications. In particular, a lack of testability, a complex stateful model, and limited influence over the generated HTML (making client-side development difficult) led developers to evaluate other options.

In response, Microsoft released the first version of ASP.NET MVC in 2009, based on the Model-View-Controller pattern, a common web design pattern used in other frameworks such as Ruby on Rails, Django, and Java Spring. This framework allowed you to separate UI elements from application logic, made testing easier, and provided tighter control over the HTML-generation process.

ASP.NET MVC has been through four more iterations since its first release, but they have all been built on the same underlying framework provided by the System.Web.dll file. This library is part of .NET Framework, so it comes pre-installed with all versions of Windows. It contains all the core code that ASP.NET uses when you build a web application.

This dependency brings both advantages and disadvantages. On the one hand, the ASP.NET framework is a reliable, battle-tested platform that's fine for building web applications on Windows. It provides a wide range of features, which have seen many years in production, and is well known by virtually all Windows web developers.

On the other hand, this reliance is limiting—changes to the underlying System.Web.dll are far-reaching and, consequently, slow to roll out. This limits the extent to which ASP.NET is free to evolve and results in release cycles only happening every few years. There's also an explicit coupling with the Windows web host, Internet Information Service (IIS), which precludes its use on non-Windows platforms.

> **NOTE** More recently, Microsoft have declared .NET Framework to be "done". It won't be being removed or replaced, but it also won't be receiving any new features. Consequently ASP.NET based on System.Web.dll will not receive new features or updates either.

In recent years, many web developers have started looking at cross-platform web frameworks that can run on Windows, as well as Linux and macOS. Microsoft felt the time had come to create a framework that was no longer tied to its Windows legacy, thus ASP.NET Core was born.

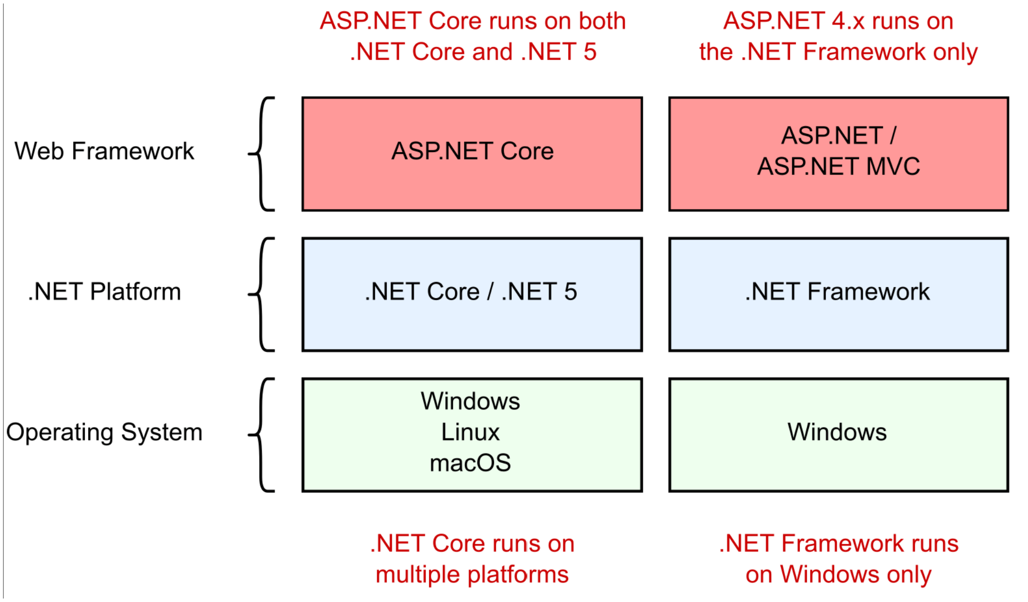
1.1.3 What is ASP.NET Core?

The development of ASP.NET Core was motivated by the desire to create a web framework with four main goals:

- To be run and developed cross-platform
- To have a modular architecture for easier maintenance
- To be developed completely as open source software
- To be applicable to current trends in web development, such as client-side applications and deploying to cloud environments

In order to achieve all these goals, Microsoft needed a platform that could provide underlying libraries for creating basic objects such as lists and dictionaries, and performing, for example, simple file operations. Up to this point, ASP.NET development had always been focused, and dependent, on the Windows-only .NET Framework. For ASP.NET Core, Microsoft created a lightweight platform that runs on Windows, Linux, and macOS called .NET Core (and subsequently .NET 5), as shown in figure 1.2.

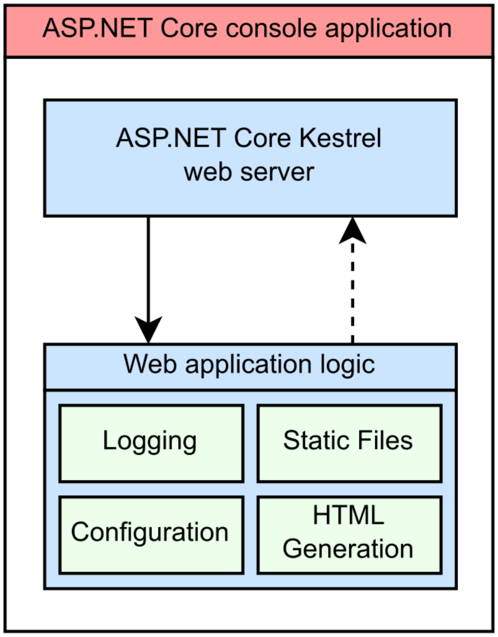
> **DEFINITION** .NET 5 is the next version of .NET Core after 3.1. It represents a unification of .NET Core and other .NET platforms into a single runtime and framework. The terms .NET Core and .NET 5 are often used interchangeably, but for consistency, I use the term .NET Core throughout this book.


ASP.NET Core runs on both .NET Core and .NET 5
ASP.NET 4.x runs on the .NET Framework only
Web Framework
ASP.NET Core
ASP.NET / ASP.NET MVC
.NET Platform
.NET Core / .NET 5
.NET Framework
Operating System
Windows
Linux
macOS
Windows
.NET Core runs on multiple platforms
.NET Framework runs on Windows only

The relationship between ASP.NET Core, ASP.NET, .NET Core, and .NET Framework. ASP.NET Core runs on .NET Core, so it can run cross-platform. Conversely, ASP.NET runs on .NET Framework only, so is tied to the Windows OS.

.NET Core shares many of the same APIs as .NET Framework, but it’s more modular, and only implements a subset of the features .NET Framework provides, with the goal of providing a simpler implementation and programming model. It’s a completely separate platform, rather than a fork of .NET Framework, though it uses similar code for many of its APIs.

With .NET Core alone, it’s possible to build console applications that run cross-platform. Microsoft created ASP.NET Core to be an additional layer on top of console applications, such that converting to a web application involves adding and composing libraries, as shown in figure 1.3.



You write a .NET Core console app that starts up an instance of an ASP.NET Core web server.

Microsoft provides a cross-platform web server by default, called Kestrel.

Your web application logic is run by Kestrel. You'll use various libraries to enable features such as logging and HTML generation as required.

The ASP.NET Core application model. The .NET Core platform provides a base console application model for running command-line apps. Adding a web server library converts this into an ASP.NET Core web app. Additional features, such as configuration and logging, are added by way of additional libraries.

By adding an ASP.NET Core web server to your .NET Core app, your application can run as a web application. ASP.NET Core is composed of many small libraries that you can choose from to provide your application with different features. You'll rarely need all the libraries available to you and you only add what you need. Some of the libraries are common and will appear in virtually every application you create, such as the ones for reading configuration files or performing logging. Other libraries build on top of these base capabilities to provide application-specific functionality, such as third-party logging-in via Facebook or Google.

Most of the libraries you'll use in ASP.NET Core can be found on GitHub, in the Microsoft ASP.NET Core organization repositories at https://github.com/dotnet/aspnetcore. You can find the core libraries here, such as the authentication and logging libraries, as well as many more peripheral libraries, such as the third-party authentication libraries.

All ASP.NET Core applications will follow a similar design for basic configuration, as suggested by the common libraries, but in general the framework is flexible, leaving you free to create your own code conventions. These common libraries, the extension libraries that build on them, and the design conventions they promote make up the somewhat nebulous term ASP.NET Core.

1.2 When to choose ASP.NET Core

Hopefully, you now have a general grasp of what ASP.NET Core is and how it was designed. But the question remains: should you use it? Microsoft is recommending that all new .NET web development should use ASP.NET Core, but switching to or learning a new web stack is a big ask for any developer or company. In this section I cover

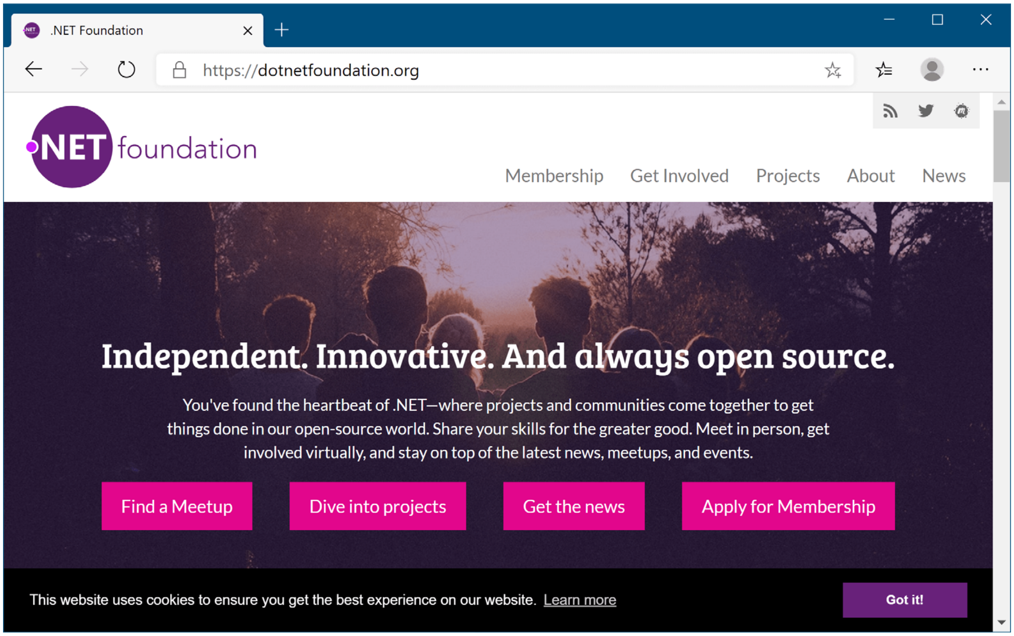
- What sort of applications you can build with ASP.NET Core
- Some of the highlights of ASP.NET Core
- Why you should consider using ASP.NET Core for new applications
- Things to consider before converting existing ASP.NET applications to ASP.NET Core

1.2.1 What type of applications can you build?

ASP.NET Core provides a generalized web framework that can be used for a variety of applications. It can most obviously be used for building rich, dynamic websites, whether they're e-commerce sites, content-based sites, or large n-tier applications—much the same as the previous version of ASP.NET.

When .NET Core was originally released, there were few third-party libraries available for building these types of complex applications. After several years of active development, that's no longer the case. Many developers have updated their libraries to work with ASP.NET Core, and many other libraries have been created to target ASP.NET Core specifically. For example, the open source content management system (CMS), Orchard[1] has been redeveloped as Orchard Core[2] to run on ASP.NET Core. In contrast, the cloudscribe[3] CMS project (figure 1.4) was written specifically for ASP.NET Core from its inception.

[1] The Orchard project (www.orchardproject.net/). Source code at https://github.com/OrchardCMS/.
[2] Orchard Core (https://www.orchardcore.net/). Source code at https://github.com/OrchardCMS/OrchardCore
[3] The cloudscribe project (https://www.cloudscribe.com/). Source code at https://github.com/cloudscribe



Independent. Innovative. And always open source.

You've found the heartbeat of .NET—where projects and communities come together to get things done in our open-source world. Share your skills for the greater good. Meet in person, get involved virtually, and stay on top of the latest news, meetups, and events.

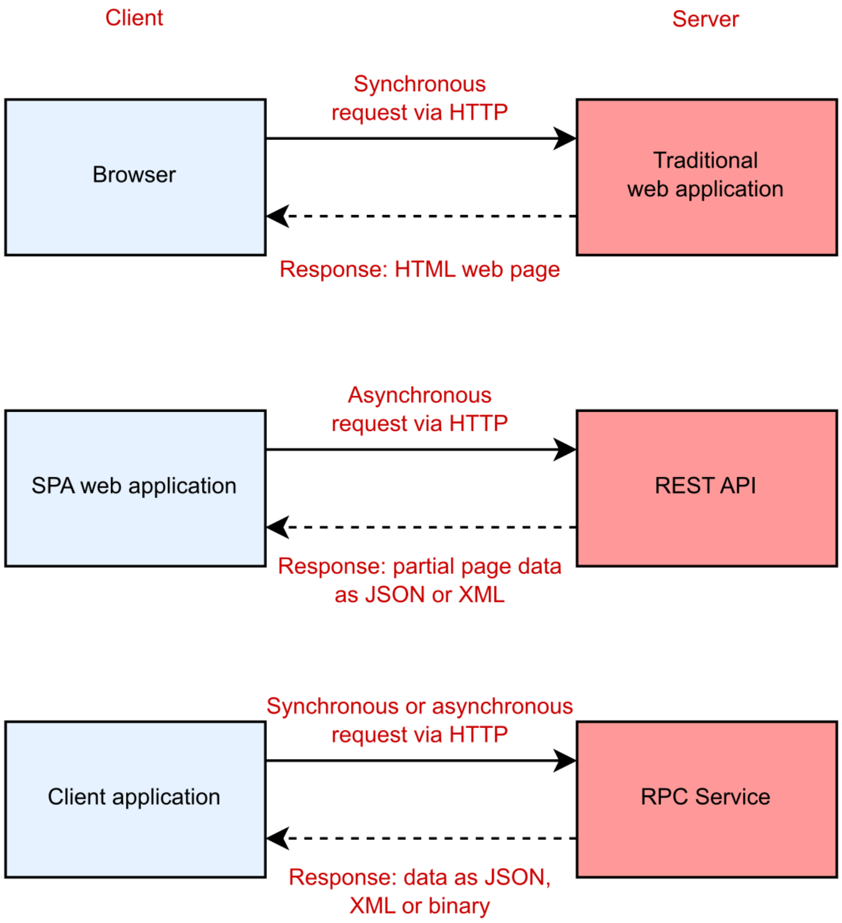
This website uses cookies to ensure you get the best experience on our website.

The .NET Foundation website (https://dotnetfoundation.org/) is build using the cloudscribe CMS and ASP.NET Core.

Traditional, page-based server-side-rendered web applications are the bread and butter of ASP.NET development, both with the previous version of ASP.NET and with ASP.NET Core. Additionally, single-page applications (SPAs), which use a client-side framework that commonly talks to a REST server, are easy to create with ASP.NET Core. Whether you're using Angular, Vue, React, or some other client-side framework, it's easy to create an ASP.NET Core application to act as the server-side API.

> **DEFINITION** ***REST*** **stands for REpresentational State Transfer. RESTful applications typically use lightweight and stateless HTTP calls to read, post (create/update), and delete data.**

ASP.NET Core isn't restricted to creating RESTful services. It's easy to create a web service or remote procedure call (RPC)-style service for your application, depending on your requirements, as shown in figure 1.5. In the simplest case, your application might expose only a single endpoint, narrowing its scope to become a microservice. ASP.NET Core is perfectly designed for building simple services thanks to its cross-platform support and lightweight design.


Client
Server
Synchronous request via HTTP
Browser
Traditional web application
Response: HTML web page
Asynchronous request via HTTP
SPA web application
REST API
Response: partial page data as JSON or XML
Synchronous or asynchronous request via HTTP
Client application
RPC Service
Response: data as JSON, XML or binary

ASP.NET Core can act as the server-side application for a variety of different clients: it can serve HTML pages for traditional web applications, it can act as a REST API for client-side SPA applications, or it can act as an ad-hoc RPC service for client applications.

You should consider multiple factors when choosing a platform, not all of which are technical. One such factor is the level of support you can expect to receive from its creators. For some organizations, this can be one of the main obstacles to adopting open source software. Luckily, Microsoft has pledged[4] to provide full support for Long Term Support (LTS) versions of .NET Core and ASP.NET Core for at least three years from the time of their release. And as all

[4] View the support policy at https://dotnet.microsoft.com/platform/support/policy/dotnet-core.

development takes place in the open, you can sometimes get answers to your questions from the general community, as well as from Microsoft directly.

When deciding whether to use ASP.NET Core, you have two primary dimensions to consider: whether you're already a .NET developer, and whether you're creating a new application or looking to convert an existing one.

1.2.2 If you're new to .NET development

If you're new to .NET development and are considering ASP.NET Core, then welcome! Microsoft is pushing ASP.NET Core as an attractive option for web development beginners, but taking .NET cross-platform means it's competing with many other frameworks on their own turf. ASP.NET Core has many selling points when compared to other cross-platform web frameworks:

- It's a modern, high-performance, open source web framework.
- It uses familiar design patterns and paradigms.
- C# is a great language (or you can use VB.NET or F# if you prefer).
- You can build and run on any platform.

ASP.NET Core is a re-imagining of the ASP.NET framework, built with modern software design principles on top of the new .NET Core platform. Although new in one sense, .NET Core has several years of widespread production use, and has drawn significantly from the mature, stable, and reliable .NET Framework, which has been used for nearly two decades. You can rest easy knowing that by choosing ASP.NET Core and .NET Core, you'll be getting a dependable platform as well as a fully-featured web framework.

Many of the web frameworks available today use similar, well-established design patterns, and ASP.NET Core is no different. For example, Ruby on Rails is known for its use of the Model-View-Controller (MVC) pattern; Node.js is known for the way it processes requests using small discrete modules (called a pipeline); and dependency injection is found in a wide variety of frameworks. If these techniques are familiar to you, you should find it easy to transfer them across to ASP.NET Core; if they're new to you, then you can look forward to using industry best practices!

NOTE You'll encounter a pipeline in chapter 3, MVC in chapter 4, and dependency injection in chapter 10.

The primary language of .NET development, and ASP.NET Core in particular, is C#. This language has a huge following, and for good reason! As an object-oriented C-based language, it provides a sense of familiarity to those used to C, Java, and many other languages. In addition, it has many powerful features, such as Language Integrated Query (LINQ), closures, and asynchronous programming constructs. The C# language is also designed in the open on GitHub, as is Microsoft's C# compiler, codenamed Roslyn.[5]

> **NOTE** I use C# throughout this book and will highlight some of the newer features it provides, but I won't be teaching the language from scratch. If you want to learn C#, I recommend *C# in Depth, fourth edition* by Jon Skeet (Manning, 2019), and *Code like a Pro in C#*, by Jort Rodenburg (Manning, 2021).

One of the major selling points of ASP.NET Core and .NET Core is the ability to develop and run on any platform. Whether you're using a Mac, Windows, or Linux, you can run the same ASP.NET Core apps and develop across multiple environments. As a Linux user, a wide range of distributions are supported (RHEL, Ubuntu, Debian, Cent-OS, Fedora, and openSUSE, to name a few), so you can be confident your operating system of choice will be a viable option. ASP.NET Core even runs on the tiny Alpine distribution, for truly compact deployments to containers.

Built with containers in mind

Traditionally, web applications were deployed directly to a server, or more recently, to a virtual machine. Virtual machines allow operating systems to be installed in a layer of virtual hardware, abstracting away the underlying hardware. This has several advantages over direct installation, such as easy maintenance, deployment, and recovery. Unfortunately, they're also heavy both in terms of file size and resource use.

This is where containers come in. Containers are far more lightweight and don't have the overhead of virtual machines. They're built in a series of layers and don't require you to boot a new operating system when starting a new one. That means they're quick to start and are great for quick provisioning. Containers, and Docker in particular, are quickly becoming the go-to platform for building large, scalable systems.

Containers have never been a particularly attractive option for ASP.NET applications, but with ASP.NET Core, .NET Core, and Docker for Windows, that's all changing. A lightweight ASP.NET Core application running on the cross-platform .NET Core framework is perfect for thin container deployments. You can learn more about your deployment options in chapter 16.

As well as running on each platform, one of the selling points of .NET is the ability to write and compile only once. Your application is compiled to Intermediate Language (IL) code, which is a platform-independent format. If a target system has the .NET Core platform installed, then you can run compiled IL from any platform. That means you can, for example, develop on a Mac or a Windows machine and deploy *the exact same files* to your production Linux machines. This compile-once, run-anywhere promise has finally been realized with ASP.NET Core and .NET Core.

[5] The C# language and .NET Compiler Platform GitHub source code repository can be found at https:// github.com/dotnet/roslyn.

1.2.3 If you’re a .NET Framework developer creating a new application

If you’re a .NET developer, then the choice of whether to invest in ASP.NET Core for new applications has largely been a question of timing. Early versions of .NET Core were lacking in some features that made it hard to adopt. With the release of .NET Core 3.1 and .NET 5, that is no longer a problem; Microsoft now explicitly advises that all new .NET applications should use .NET Core. Microsoft has pledged to provide bug and security fixes for the older ASP.NET framework, but it won’t receive any more feature updates. .NET Framework isn’t being removed, so your old applications will continue to work, but you shouldn’t use it for new development.

The main benefits of ASP.NET Core over the previous ASP.NET framework are:

- Cross-platform development and deployment
- A focus on performance as a feature
- A simplified hosting model
- Regular releases with a shorter release cycle
- Open source
- Modular features

As a .NET developer, if you aren’t using any Windows-specific constructs, such as the Registry, then the ability to build and deploy applications cross-platform opens the door to a whole new avenue of applications: take advantage of cheaper Linux VM hosting in the cloud, use Docker containers for repeatable continuous integration, or write .NET code on your Mac without needing to run a Windows virtual machine. ASP.NET Core, in combination with .NET Core, makes all this possible.

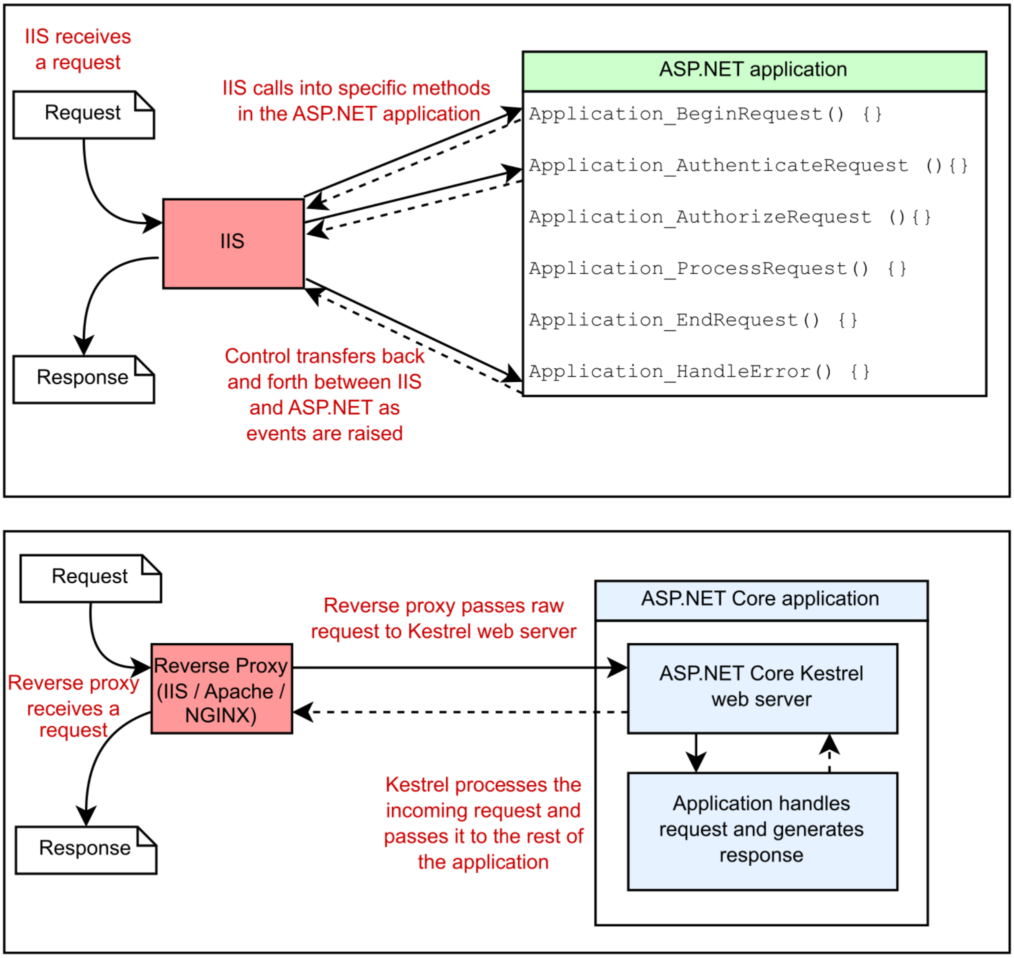
.NET Core is inherently cross-platform, but you can still use platform-specific features if you need to. For example, Windows-specific features like the Registry or Directory Services can be enabled with a compatibility pack[6] that makes these APIs available in .NET Core. They’re only available when running .NET Core on Windows, not on Linux or macOS, so you need to take care that such applications only run in a Windows environment, or account for the potential missing APIs.

The hosting model for the previous ASP.NET framework was a relatively complex one, relying on Windows IIS to provide the web server hosting. In a cross-platform environment, this kind of symbiotic relationship isn’t possible, so an alternative hosting model has been adopted, one which separates web applications from the underlying host. This opportunity has led to the development of Kestrel: a fast, cross-platform HTTP server on which ASP.NET Core can run.

Instead of the previous design, whereby IIS calls into specific points of your application, ASP.NET Core applications are console applications that self-host a web server and handle

[6] The Windows Compatibility Pack is designed to help port code from .NET Framework to .NET Core. See http://mng.bz/50hu.

requests directly, as shown in figure 1.6. This hosting model is conceptually much simpler and allows you to test and debug your applications from the command line, though it doesn’t remove the need to run IIS (or equivalent) in production, as you’ll see in section 1.3.


IIS receives a request
Request
IIS calls into specific methods in the ASP.NET application
ASP.NET application
Application_BeginRequest() {}
Application_AuthenticateRequest (){}
Application_AuthorizeRequest (){}
Application_ProcessRequest() {}
Application_EndRequest() {}
Application_HandleError() {}
IIS
Response
Control transfers back and forth between IIS and ASP.NET as events are raised
Request
Reverse proxy passes raw request to Kestrel web server
ASP.NET Core application
Reverse proxy receives a request
Reverse Proxy (IIS / Apache / NGINX)
ASP.NET Core Kestrel web server
Kestrel processes the incoming request and passes it to the rest of the application
Application handles request and generates response
Response

The difference between hosting models in ASP.NET (top) and ASP.NET Core (bottom). With the previous version of ASP.NET, IIS is tightly coupled with the application. The hosting model in ASP.NET Core is simpler; IIS hands off the request to a self-hosted web server in the ASP.NET Core application and receives the response, but has no deeper knowledge of the application.

Changing the hosting model to use a built-in HTTP web server has created another opportunity. Performance has been somewhat of a sore point for ASP.NET applications in the past. It’s certainly possible to build high-performing applications—Stack Overflow (https://stackoverflow.com) is testament to that—but the web framework itself isn't designed with performance as a priority, so it can end up being somewhat of an obstacle.

To be competitive cross-platform, the ASP.NET team have focused on making the Kestrel HTTP server as fast as possible. TechEmpower (www.techempower.com/benchmarks) has been running benchmarks on a whole range of web frameworks from various languages for several years now. In Round 19 of the plain text benchmarks, TechEmpower announced that ASP.NET Core with Kestrel was the fastest of over 400 frameworks tested![7]

Web servers: naming things is hard

One of the difficult aspects of programming for the web is the confusing array of often conflicting terminology. For example, if you've used IIS in the past, you may have described it as a web server, or possibly a web host. Conversely, if you've ever built an application using Node.js, you may have also referred to that application as a web server. Alternatively, you may have called the physical machine on which your application runs a web server!

Similarly, you may have built an application for the internet and called it a website or a web application, probably somewhat arbitrarily based on the level of dynamism it displayed.

In this book, when I say "web server" in the context of ASP.NET Core, I am referring to the HTTP server that runs as part of your ASP.NET Core application. By default, this is the Kestrel web server, but that's not a requirement. It would be possible to write a replacement web server and substitute it for Kestrel if you desired.

The web server is responsible for receiving HTTP requests and generating responses. In the previous version of ASP.NET, IIS took this role, but in ASP.NET Core, Kestrel is the web server.

I will only use the term web application to describe ASP.NET Core applications in this book, regardless of whether they contain only static content or are completely dynamic. Either way, they're applications that are accessed via the web, so that name seems the most appropriate!

Many of the performance improvements made to Kestrel did not come from the ASP.NET team themselves, but from contributors to the open source project on GitHub.[8] Developing in the open means you typically see fixes and features make their way to production faster than you would for the previous version of ASP.NET, which was dependent on .NET Framework and Windows and, as such, had long release cycles.

In contrast, .NET Core, and hence ASP.NET Core, is designed to be released in small increments. Major versions will be released on a predictable cadence, with a new version every year, and a new Long Term Support (LTS) version released every two years[9]. In addition, bug fixes and minor updates can be released as and when they're needed. Additional functionality is provided as NuGet packages, independent of the underlying .NET Core platform.

[7] As always in web development, technology is in a constant state of flux, so these benchmarks will evolve over time. Although ASP.NET Core may not maintain its top ten slot, you can be sure that performance is one of the key focal points of the ASP.NET Core team.

[8] The Kestrel HTTP server GitHub project can be found in the ASP.NET Core repository at https://github.com/dotnet/aspnetcore.

[9] The release schedule for .NET 5 and beyond: https://devblogs.microsoft.com/dotnet/introducing-net-5/

NOTE NuGet is a package manager for .NET that enables importing libraries into your projects. It's equivalent to Ruby Gems, npm for JavaScript, or Maven for Java.

To enable this, ASP.NET Core is highly modular, with as little coupling to other features as possible. This modularity lends itself to a pay-for-play approach to dependencies, where you start from a bare-bones application and only add the additional libraries you require, as opposed to the kitchen-sink approach of previous ASP.NET applications. Even MVC is an optional package! But don't worry, this approach doesn't mean that ASP.NET Core is lacking in features; it means you need to opt in to them. Some of the key infrastructure improvements include

- Middleware "pipeline" for defining your application's behavior
- Built-in support for dependency injection
- Combined UI (MVC) and API (Web API) infrastructure
- Highly extensible configuration system
- Scalable for cloud platforms by default using asynchronous programming

Each of these features was possible in the previous version of ASP.NET but required a fair amount of additional work to set up. With ASP.NET Core, they're all there, ready, and waiting to be connected!

Microsoft fully supports ASP.NET Core, so if you have a new system you want to build, then there's no significant reason not to. The largest obstacle you're likely to come across is when you want to use programming models that are no longer supported in ASP.NET Core, such as Web Forms or WCF server, as I'll discuss in the next section.

Hopefully, this section has whetted your appetite with some of the many reasons to use ASP.NET Core for building new applications. But if you're an existing ASP.NET developer considering whether to convert an existing ASP.NET application to ASP.NET Core, that's another question entirely.

1.2.4 Converting an existing ASP.NET application to ASP.NET Core

In contrast with new applications, an existing application is presumably already providing value, so there should always be a tangible benefit to performing what may amount to a significant rewrite in converting from ASP.NET to ASP.NET Core. The advantages of adopting ASP.NET Core are much the same as for new applications: cross-platform deployment, modular features, and a focus on performance. Determining whether the benefits are sufficient will depend largely on the particulars of your application, but there are some characteristics that are clear indicators *against* conversion:

- Your application uses ASP.NET Web Forms
- Your application is built using WCF
- Your application is large, with many "advanced" MVC features

If you have an ASP.NET Web Forms application, then attempting to convert it to ASP.NET Core isn't advisable. Web Forms is inextricably tied to System.Web.dll, and as such will likely never be available in ASP.NET Core. Converting an application to ASP.NET Core would effectively involve rewriting the application from scratch, not only shifting frameworks but also shifting design paradigms. A better approach would be to slowly introduce Web API concepts and try to reduce the reliance on legacy Web Forms constructs such as ViewData. You can find many resources online to help you with this approach, in particular, the www.asp.net/web-api website.

Windows Communication Foundation (WCF) is only partially supported in ASP.NET Core.[10] It's possible to consume some WCF services, but support is spotty at best. There's no supported way to host a WCF service from an ASP.NET Core application, so if you absolutely must support WCF, then ASP.NET Core may be best avoided for now.

> **TIP** If you like WCFs RPC-style of programming, but don't have a hard requirement on WCF itself, then consider using gRPC instead. gRPC is a modern RPC framework with many similar concepts as WCF and is supported by ASP.NET Core out-of-the-box[11].

If your existing application is complex and makes extensive use of the previous MVC or Web API extensibility points or message handlers, then porting your application to ASP.NET Core may be more difficult. ASP.NET Core is built with many similar features to the previous version of ASP.NET MVC, but the underlying architecture is different. Several of the previous features don't have direct replacements, and so will require rethinking.

The larger the application, the greater the difficulty you're likely to have converting your application to ASP.NET Core. Microsoft itself suggests that porting an application from ASP.NET MVC to ASP.NET Core is at least as big a rewrite as porting from ASP.NET Web Forms to ASP.NET MVC. If that doesn't scare you, then nothing will!

If an application is rarely used, isn't part of your core business, or won't need significant development in the near term, then I strongly suggest you *don't* try to convert it to ASP.NET Core. Microsoft will support .NET Framework for the foreseeable future (Windows itself depends on it!) and the payoff in converting these "fringe" applications is unlikely to be worth the effort.

So, when *should* you port an application to ASP.NET Core? As I've already mentioned, the best opportunity for getting started is on small, green-field, new projects instead of existing applications. That said, if the existing application in question is small, or will need significant future development, then porting may be a good option. It is always best to work in small iterations where possible, rather than attempting to convert the entire application at once. But if your application consists primarily of MVC or Web API controllers and associated Razor views, then moving to ASP.NET Core may well be a good choice.

[10] You can find the client libraries for using WCF with .NET Core at https://github.com/dotnet/wcf.

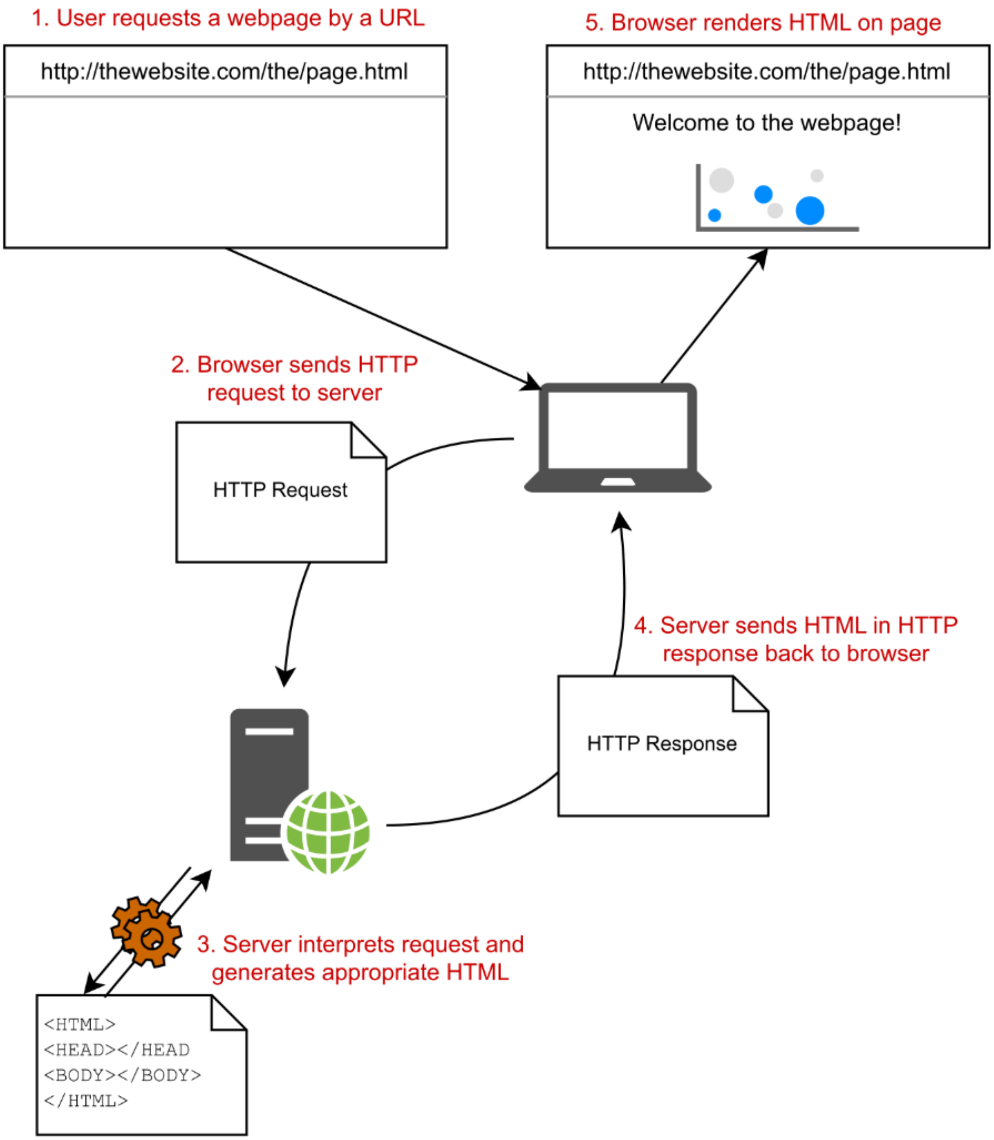
[11] You can find an eBook from Microsoft on gRPC for WCF developers at https://docs.microsoft.com/en-us/dotnet/architecture/grpc-for-wcf-developers/.

1.3 How does ASP.NET Core work?

By now, you should have a good idea of what ASP.NET Core is and the sort of applications you should use it for. In this section, you’ll see how an application built with ASP.NET Core works, from the user requesting a URL, to a page being displayed on the browser. To get there, first you’ll see how an HTTP request works for any web server, and then you’ll see how ASP.NET Core extends the process to create dynamic web pages.

1.3.1 How does an HTTP web request work?

As you know, ASP.NET Core is a framework for building web applications that serve data from a server. One of the most common scenarios for web developers is building a web app that you can view in a web browser. The high-level process you can expect from any web server is shown in figure 1.7.


1. User requests a webpage by a URL
http://thewebsite.com/the/page.html
5. Browser renders HTML on page
http://thewebsite.com/the/page.html
Welcome to the webpage!
2. Browser sends HTTP request to server
HTTP Request
4. Server sends HTML in HTTP response back to browser
HTTP Response
3. Server interprets request and generates appropriate HTML
<HTML>
<HEAD></HEAD
<BODY></BODY>
</HTML>

Requesting a web page. The user starts by requesting a web page, which causes an HTTP request to be sent to the server. The server interprets the request, generates the necessary HTML, and sends it back in an HTTP response. The browser can then display the web page.

The process begins when a user navigates to a website or types a URL in their browser. The URL or web address consists of a *hostname* and a *path* to some resource on the web app.

Navigating to the address in your browser sends a request from the user's computer to the server on which the web app is hosted, using the HTTP protocol.

> **DEFINITION** The *hostname* of a website uniquely identifies its location on the internet by mapping via the Domain Name Service (DNS) to an IP Address. Examples include microsoft.com, www.google.co.uk, and facebook.com.

The request passes through the internet, potentially to the other side of the world, until it finally makes its way to the server associated with the given hostname on which the web app is running. The request is potentially received and rebroadcast at multiple routers along the way, but it's only when it reaches the server associated with the hostname that the request is processed.

Once the server receives the request, it will check that it makes sense, and if it does, will generate an HTTP response. Depending on the request, this response could be a web page, an image, a JavaScript file, or a simple acknowledgment. For this example, I'll assume the user has reached the homepage of a web app, and so the server responds with some HTML. The HTML is added to the HTTP response, which is then sent back across the internet to the browser that made the request.

As soon as the user's browser begins receiving the HTTP response, it can start displaying content on the screen, but the HTML page may also reference other pages and links on the server. To display the complete web page, instead of a static, colorless, raw HTML file, the browser must repeat the request process, fetching every referenced file. HTML, images, CSS for styling, and JavaScript files for extra behavior are all fetched using the exact same HTTP request process.

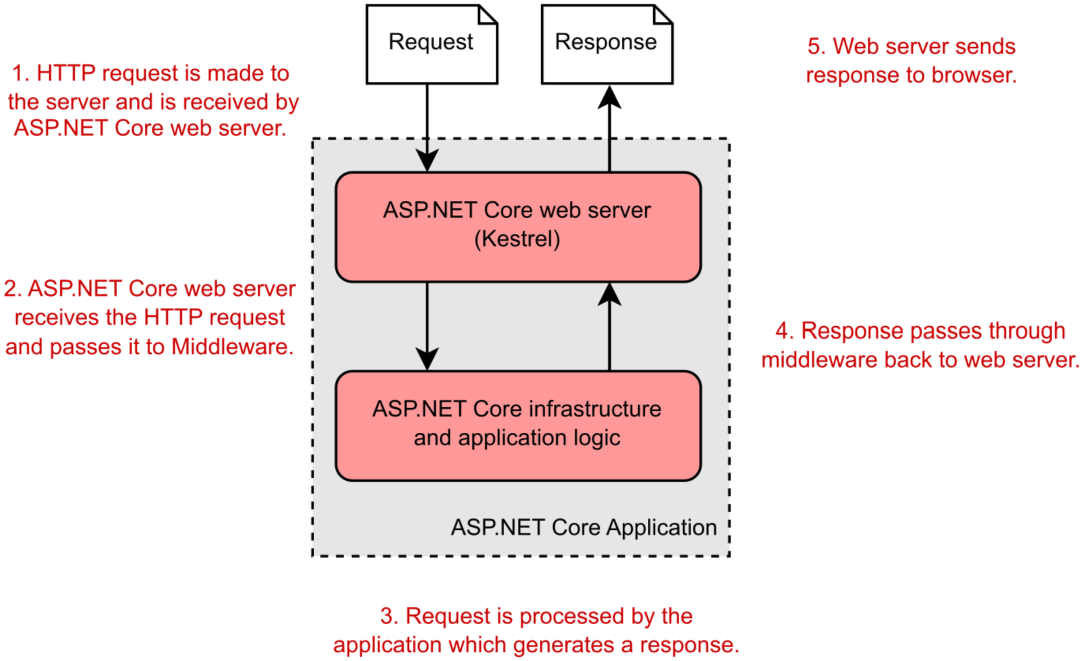
Pretty much all interactions that take place on the internet are a facade over this same basic process. A basic web page may only require a few simple requests to fully render, whereas a modern, large web page may take hundreds. The Amazon.com homepage (www.amazon.com), for example, makes 606 requests, including 3 CSS files, 12 JavaScript files, and 402 image files!

Now you have a feel for the process, let's see how ASP.NET Core dynamically generates the response on the server.

1.3.2 How does ASP.NET Core process a request?

When you build a web application with ASP.NET Core, browsers will still be using the same HTTP protocol as before to communicate with your application. ASP.NET Core itself encompasses everything that takes place on the server to handle a request, including verifying the request is valid, handling login details, and generating HTML.

Just as with the generic web page example, the request process starts when a user's browser sends an HTTP request to the server, as shown in figure 1.8.


Request
Response
1. HTTP request is made to the server and is received by ASP.NET Core web server.
5. Web server sends response to browser.
ASP.NET Core web server (Kestrel)
2. ASP.NET Core web server receives the HTTP request and passes it to Middleware.
4. Response passes through middleware back to web server.
ASP.NET Core infrastructure and application logic
ASP.NET Core Application
3. Request is processed by the application which generates a response.

How an ASP.NET Core application processes a request. A request is received by the ASP.NET Core application, which runs a self-hosted web server. The web server processes the request and passes it to the body of the application, which generates a response and returns it to the web server. The web server sends this response to the browser.

The request is received from the network by your ASP.NET Core application. Every ASP.NET Core application has a built-in web server, Kestrel by default, which is responsible for receiving raw requests and constructing an internal representation of the data, an `HttpContext` object, which can be used by the rest of the application.

From this representation, your application should have all the details it needs to create an appropriate response to the request. It can use the details stored in `HttpContext` to generate an appropriate response, which may be to generate some HTML, to return an "access denied" message, or to send an email, all depending on your application's requirements.

Once the application has finished processing the request, it will return the response to the web server. The ASP.NET Core web server will convert the representation into a raw HTTP response and send it to the network, which will forward it to the user's browser.

To the user, this process appears to be the same as for the generic HTTP request shown in figure 1.7—the user sent an HTTP request and received an HTTP response. All the differences are server-side, within our application.

ASP.NET Core and Reverse Proxies

You can expose ASP.NET Core applications directly to the internet, so that Kestrel receives requests directly from the network. However, it's more common to use a reverse proxy between the raw network and your application. In Windows, the reverse-proxy server will typically be IIS, and on Linux or macOS it might be NGINX, HAProxy or Apache.

A *reverse proxy* is software responsible for receiving requests and forwarding them to the appropriate web server. The reverse proxy is exposed directly to the internet, whereas the underlying web server is exposed only to the proxy. This setup has several benefits, primarily security and performance for the web servers.

You may be thinking that having a reverse proxy *and* a web server is somewhat redundant. Why not have one or the other? Well, one of the benefits is the decoupling of your application from the underlying operating system. The same ASP.NET Core web server, Kestrel, can be cross-platform and used behind a variety of proxies without putting any constraints on a particular implementation. Alternatively, if you wrote a new ASP.NET Core web server, you could use that in place of Kestrel without needing to change anything else about your application.

Another benefit of a reverse proxy is that it can be hardened against potential threats from the public internet. They're often responsible for additional aspects, such as restarting a process that has crashed. Kestrel can stay as a simple HTTP server without having to worry about these extra features when it's used behind a reverse proxy. Think of it as a simple separation of concerns: Kestrel is concerned with generating HTTP responses; a reverse proxy is concerned with handling the connection to the internet.

You've seen how requests and responses find their way to and from an ASP.NET Core application, but I haven't yet touched on how the response is generated. In part 1 of this book, we'll look at the components that make up a typical ASP.NET Core application and how they all fit together. A lot goes into generating a response in ASP.NET Core, typically all within a fraction of a second, but over the course of the book we'll step through an application slowly, covering each of the components in detail.

1.4 What you will learn in this book

This book takes you on an in-depth tour of the ASP.NET Core framework. To benefit from the book, you should be familiar with C# or a similar objected-oriented language. Basic familiarity with web concepts like HTML and JavaScript will also be beneficial. You will learn:

- How to create page-based applications with Razor Pages.
- Key ASP.NET Core concepts like model-binding, validation, and routing.
- How to generate HTML for web pages using Razor syntax and Tag Helpers.
- To use features like dependency injection, configuration, and logging as your applications grow more complex.
- How to protect your application using security best practices.

Throughout the book we'll use a variety of examples to learn and explore concepts. The examples are generally small and self-contained, so we can focus on a single feature at a time.

I'll be using Visual Studio for most of the examples in this book, but you'll be able to follow along using your favorite editor or IDE.

TIP You can install .NET Core from https://get.asp.net/. Appendix A contains further details on how to configure your development environment for working with .NET Core.

In the next chapter, you'll create your first application from a template and run it. We'll walk through each of the main components that make up your application and see how they all work together to render a web page.

1.5 Summary

- ASP.NET Core is a new web framework built with modern software architecture practices and modularization as its focus.
- It's best used for new, "green-field" projects.
- Legacy technologies such as WCF Server and Web Forms can't be used with ASP.NET Core.
- ASP.NET Core runs on the cross-platform .NET Core platform. You can access Windows-specific features such as the Windows Registry by using the Windows Compatibility Pack.
- Fetching a web page involves sending an HTTP request and receiving an HTTP response.
- ASP.NET Core allows dynamically building responses to a given request.
- An ASP.NET Core application contains a web server, which serves as the entry-point for a request.
- ASP.NET Core apps are typically protected from the internet by a reverse-proxy server, which forwards requests to the application.

2
Your first application

This chapter covers

- **Creating your first ASP.NET Core web application**
- **Running your application**
- **Understanding the components of your application**

After reading chapter 1, you should have a general idea of how ASP.NET Core applications work and when you should use them. You should have also set up a development environment to start building applications. In this chapter, you'll dive right in by creating your first web app. You'll get to kick the tires and poke around a little to get a feel for how it works, and in later chapters, I'll show how you go about customizing and building your own applications.

As you work through this chapter, you should begin to get a grasp of the various components that make up an ASP.NET Core application, as well as an understanding of the general application-building process. Most applications you create will start from a similar *template*, so it's a good idea to get familiar with the setup as soon as possible.

> **DEFINITION** A *template* provides the basic code required to build an application. You can use a template as the starting point for building your own apps.

I'll start by showing how to create a basic ASP.NET Core application using one of the Visual Studio templates. If you're using other tooling, such as the .NET CLI, then you'll have similar templates available. I use Visual Studio 2019 and ASP.NET Core 3.1 in this chapter, but I also provide tips for working with the .NET CLI.

> **TIP** You can view the application code for this chapter in the GitHub repository for the book at https://github.com/andrewlock/asp-dot-net-core-in-action-2e.

Once you’ve created your application, I’ll show you how to restore all the necessary dependencies, compile your application, and run it to see the HTML output. The application will be simple in some respects—it will only have two different pages—but it’ll be a fully configured ASP.NET Core application.

Having run your application, the next step is to understand what’s going on! We’ll take a journey through all the major parts of an ASP.NET Core application, looking at how to configure the web server, the middleware pipeline, and HTML generation, among other things. We won’t go into detail at this stage, but you’ll get a feel for how they all work together to create a complete application.

We’ll begin by looking at the plethora of files created when you start a new project and learn how a typical ASP.NET Core application is laid out. In particular, I’ll focus on the Program.cs and Startup.cs files. Virtually the entire configuration of your application takes place in these two files, so it’s good to get to grips with what they are for and how they’re used. You’ll see how to define the middleware pipeline for your application, and how you can customize it.

Finally, you’ll see how the app generates HTML in response to a request, looking at each of the components that make up the Razor Pages endpoint. You’ll see how it controls what code is run in response to a request, and how to define the HTML that should be returned for a particular request.

At this stage, don’t worry if you find parts of the project confusing or complicated; you’ll be exploring each section in detail as you move through the book. By the end of the chapter, you should have a basic understanding of how ASP.NET Core applications are put together, right from when your application is first run to when a response is generated. Before we begin though, we’ll review how ASP.NET Core applications handle requests.

2.1 A brief overview of an ASP.NET Core application

In chapter 1, I described how a browser makes an HTTP request to a server and receives a response, which it uses to render HTML on the page. ASP.NET Core allows you to dynamically generate that HTML depending on the particulars of the request so that you can, for example, display different data depending on the current logged-in user.

Say you want to create a web app to display information about your company. You could create a simple ASP.NET Core app to achieve this, especially if you might later want to add dynamic features to your app. Figure 2.1 shows how the application would handle a request for a page in your application.

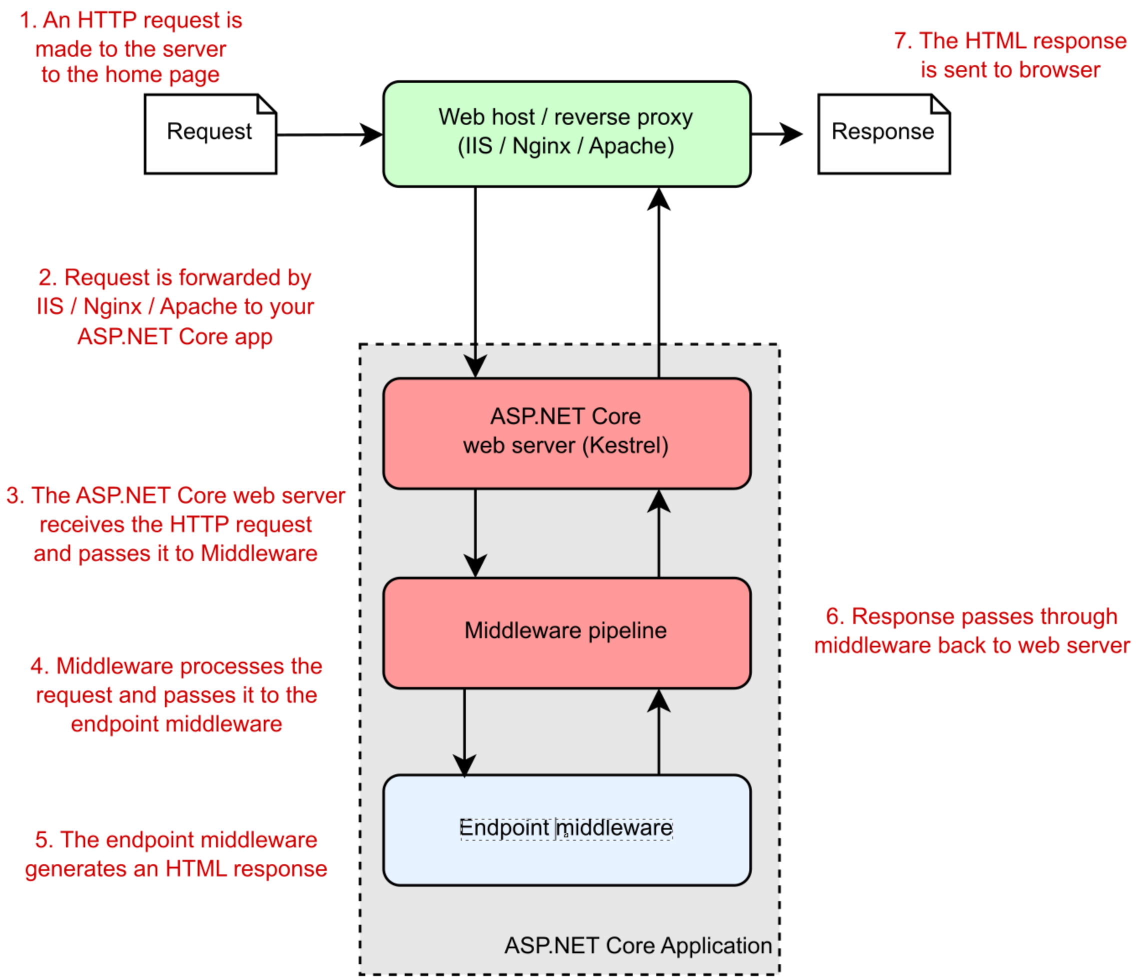

1. An HTTP request is made to the server to the home page
Request
Web host / reverse proxy (IIS / Nginx / Apache)
Response
7. The HTML response is sent to browser
2. Request is forwarded by IIS / Nginx / Apache to your ASP.NET Core app
ASP.NET Core web server (Kestrel)
3. The ASP.NET Core web server receives the HTTP request and passes it to Middleware
Middleware pipeline
6. Response passes through middleware back to web server
4. Middleware processes the request and passes it to the endpoint middleware
Endpoint middleware
5. The endpoint middleware generates an HTML response
ASP.NET Core Application

Figure 2.1 An overview of an ASP.NET Core application. The ASP.NET Core application contains of a number of blocks that process an incoming request from the browser. Every request passes to the middleware pipeline. It potentially modifies it, then passes it to the endpoint middleware at the end of the pipeline to generate a response. The response passes back through the middleware, to the server, and finally, out to the browser.

Much of this diagram should be familiar to you from figure 1.8 in chapter 1; the request and response, the reverse proxy, and the ASP.NET Core web server are all still there, but you'll notice that I've expanded the ASP.NET Core application itself to show the middleware pipeline and the endpoint middleware. This is the main custom part of your app that goes into generating the response from a request.

The first port of call after the reverse proxy forwards a request is the ASP.NET Core web server, which is the default cross-platform Kestrel server. Kestrel takes the raw incoming network request and uses it to generate an `HttpContext` object that the rest of the application can use.

The HttpContext object

The `HttpContext` constructed by the ASP.NET Core web server is used by the application as a sort of storage box for a single request. Anything that's specific to this particular request and the subsequent response can be associated with it and stored in it. This could include properties of the request, request-specific services, data that's been loaded, or errors that have occurred. The web server fills the initial `HttpContext` with details of the original HTTP request and other configuration details and passes it on to the rest of the application.

NOTE Kestrel isn't the only HTTP server available in ASP.NET Core, but it's the most performant and is cross-platform. I'll only refer to Kestrel throughout the book. The main alternative, HTTP.sys, only runs on Windows and can't be used with IIS.[12]

Kestrel is responsible for receiving the request data and constructing a C# representation of the request, but it doesn't attempt to generate a response directly. For that, Kestrel hands the `HttpContext` to the middleware pipeline found in every ASP.NET Core application. This is a series of components that processes the incoming request to perform common operations such as logging, handling exceptions, or serving static files.

NOTE You'll learn about the middleware pipeline in detail in the next chapter.

At the end of the middleware pipeline is the *endpoint* middleware. This middleware is responsible for calling the code that generates the final response. In most applications that will be an MVC or Razor Pages block.

Razor Pages are responsible for generating the HTML that makes up the pages of a typical ASP.NET Core web app. They're also typically where you find most of the business logic of your app, by calling out to various services in response to the data contained in the original request. Not every app needs an MVC or Razor Pages block, but it's typically how you'll build most apps that display HTML to a user.

NOTE I'll cover Razor Pages and MVC controllers in chapter 4, including how to choose between them. I cover generating HTML in chapters 7 and 8.

[12] If you want to learn more about HTTP.sys, the documentation describes the server and how to use it: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/httpsys.

Most ASP.NET Core applications follow this basic architecture, and the example in this chapter is no different. First, you'll see how to create and run your application, then we'll look at how the code corresponds to the outline in figure 2.1. Without further ado, let's create an application!

2.2 Creating your first ASP.NET Core application

You can start building applications with ASP.NET Core in many different ways, depending on the tools and operating system you're using. Each set of tools will have slightly different templates, but they have many similarities. The example used throughout this chapter is based on a Visual Studio 2019 template, but you can easily follow along with templates from the .NET CLI or Visual Studio for Mac.

> **REMINDER** This chapter uses Visual Studio 2019 and ASP.NET Core 3.1.

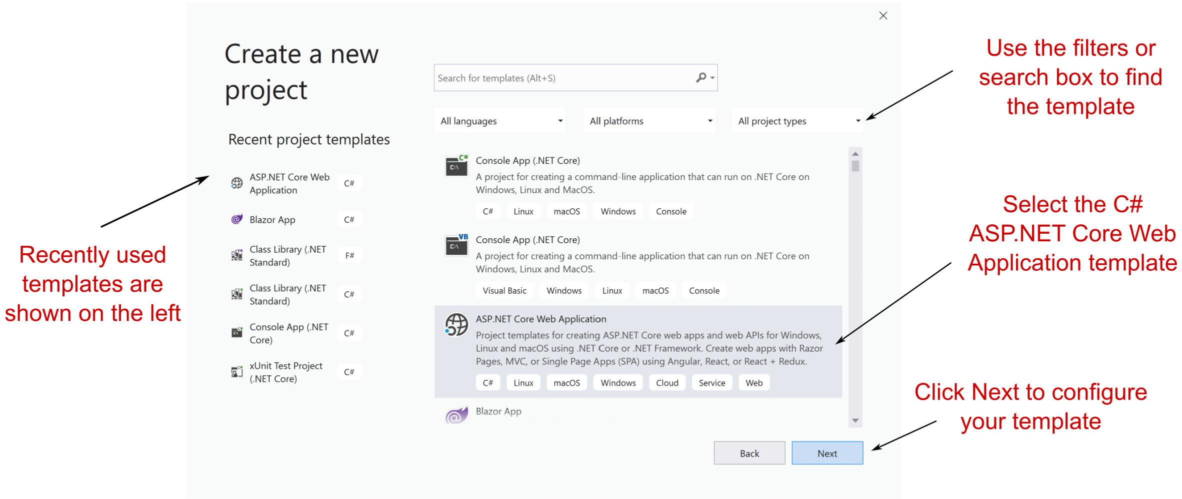
Getting an application up and running typically follows four basic steps, which we'll work through in this chapter:

1. *Generate*—Create the base application from a template to get started.
2. *Restore*—Restore all the packages and dependencies to the local project folder using NuGet.
3. *Build*—Compile the application and generate all the necessary assets.
4. *Run*—Run the compiled application.

2.2.1 Using a template to get started

Using a template can quickly get you up and running with an application, automatically configuring many of the fundamental pieces for you. Both Visual Studio and the .NET CLI come with a number of standard templates for building web applications, console applications, and class libraries. To create your first web application, open Visual Studio and perform the following steps:

1. Choose Create a New Project from the splash screen, or choose File > New > Project from the main Visual Studio screen..
2. From the list of templates, choose ASP.NET Core Web Application, ensuring you select the C# language template, as shown in figure 2.2. Click Next



Create a new project

Search for templates (Alt+S)

All languages | All platforms | All project types

Recent project templates

- ASP.NET Core Web Application — C#
- Blazor App — C#
- Class Library (.NET Standard) — F#
- Class Library (.NET Standard) — C#
- Console App (.NET Core) — C#
- xUnit Test Project (.NET Core) — C#

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
C# | Linux | macOS | Windows | Console

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
Visual Basic | Windows | Linux | macOS | Console

ASP.NET Core Web Application
Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Angular, React, or React + Redux.
C# | Linux | macOS | Windows | Cloud | Service | Web

Blazor App

Use the filters or search box to find the template

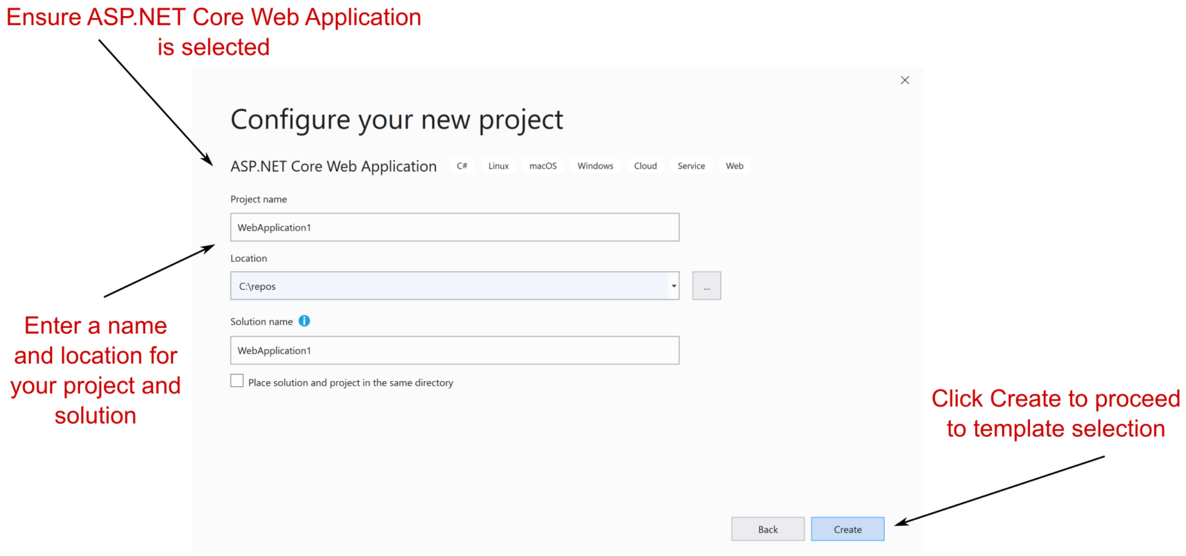
Recently used templates are shown on the left

Select the C# ASP.NET Core Web Application template

Click Next to configure your template

Figure 2.2 The new project dialog. Select the C# ASP.NET Core Web Application template from the list on the right-hand side. When you next create a new project, you can select from the recent templates list on the left.

3. On the next screen, enter a project name, Location, and a solution name, and click Create, as shown in figure 2.3.



Ensure ASP.NET Core Web Application is selected

Configure your new project

ASP.NET Core Web Application — C# | Linux | macOS | Windows | Cloud | Service | Web

Project name: WebApplication1

Location: C:\repos

Solution name: WebApplication1

☐ Place solution and project in the same directory

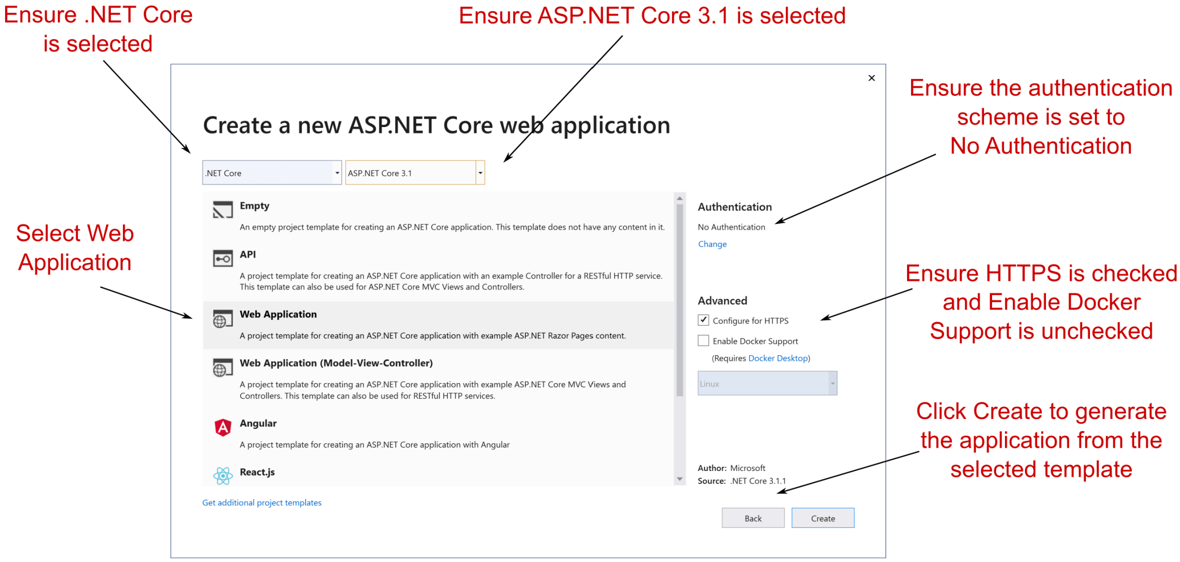
Enter a name and location for your project and solution

Click Create to proceed to template selection

Figure 2.3 The configure your new project dialog. To create a new .NET Core application, select ASP.NET Core Web Application from the template screen. On the following screen, enter a project name, location, and a solution name and click Create.

4. On the following screen (figure 2.4):

- Ensure .NET Core is selected.
- Select ASP.NET Core 3.1. The generated application will target ASP.NET Core 3.1.
- Select Web Application. This ensures you create a Razor Pages web application that generates HTML and is designed to be viewed by users in a web browser directly. The alternative Web Application (Model-View-Controller) template uses traditional MVC controllers instead of Razor Pages. The API template generates an application that returns data in a format that can be consumed by single-page applications (SPAs) and APIs. The Angular, React.js, and React.js and Redux templates create applications for specific SPAs.
- Ensure No Authentication is specified. You’ll learn how to add users to your app in chapter 14.
- Ensure Configure for HTTPS is checked
- Ensure Enable Docker Support is unchecked.
- Click Create.



Ensure .NET Core is selected

Ensure ASP.NET Core 3.1 is selected

Ensure the authentication scheme is set to No Authentication

Select Web Application

Ensure HTTPS is checked and Enable Docker Support is unchecked

Click Create to generate the application from the selected template

Create a new ASP.NET Core web application

.NET Core | ASP.NET Core 3.1

Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Web Application
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

Web Application (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Angular
A project template for creating an ASP.NET Core application with Angular

React.js

Authentication
No Authentication

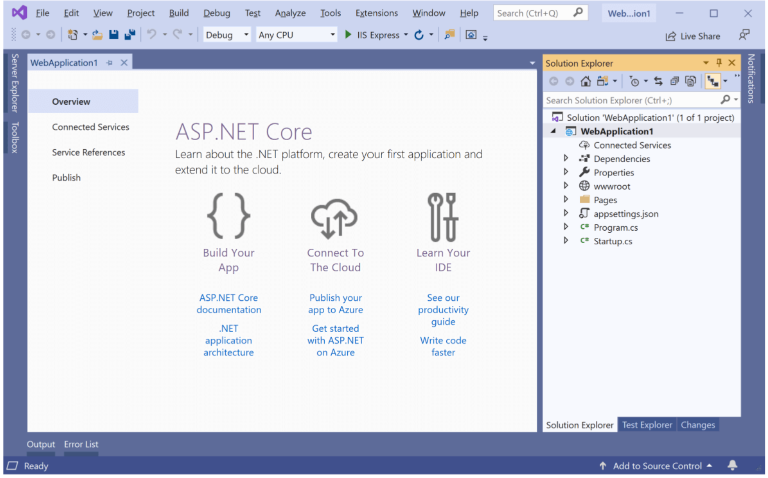
Advanced
☑ Configure for HTTPS
□ Enable Docker Support
(Requires Docker Desktop)

Author: Microsoft
Source: .NET Core 3.1.1

Figure 2.4 The web application template screen. This screen follows on from the configure your project dialog and lets you customize the template that will generate your application. For this starter project, you’ll create a Razor Pages web application without authentication.

5. Wait for Visual Studio to generate the application from the template. Once Visual Studio has finished, you’ll be presented with an introductory page about ASP.NET Core, and

you should be able to see that Visual Studio has created and added a number of files to your project, as shown in figure 2.5.



Learn about the .NET platform, create your first application and extend it to the cloud.

An introductory page is shown when your project is first created

Solution Explorer shows the files in your project

Figure 2.5 Visual Studio after creating a new ASP.NET Core application from a template. The Solution Explorer shows your newly created project. The introductory page has helpful links for learning about ASP.NET Core.

NOTE If you're developing using the .NET CLI, you can create a similar application by running `dotnet new webapp -o WebApplication1` from the command line. The `-o` argument ensures the CLI creates the template in a subfolder called WebApplication1.

2.2.2 Building the application

At this point, you have most of the files necessary to run your application, but you've got two steps left. First, you need to ensure all the dependencies used by your project are copied to your local directory, and second, you need to compile your application so that it can be run.

The first of these steps isn't strictly necessary, as both Visual Studio and the .NET CLI automatically restore packages when they first create your project, but it's good to know what's going on. In earlier versions of the .NET CLI, before 2.0, you needed to manually restore packages using `dotnet restore`.

You can compile your application by choosing Build > Build Solution, by using the shortcut Ctrl+Shift+B, or by running `dotnet build` from the command line. If you build from Visual Studio, the output window shows the progress of the build, and assuming everything is hunky dory, will compile your application, ready for running. You can also run the `dotnet build` console commands from the Package Manager Console in Visual Studio.

TIP Visual Studio and the .NET CLI tools will automatically build your application when you run it if they detect that a file has changed, so you generally won't need to explicitly perform this step yourself.

NuGet packages and the .NET Core command line interface

One of the foundational components of .NET Core cross-platform development is the .NET Core command line interface (CLI). This provides several basic commands for creating, building, and running .NET Core applications. Visual Studio effectively calls these automatically, but you can also invoke them directly from the command line if you're using a different editor. The most common commands during development are

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

Each of these commands should be run inside your project folder and will act on that project alone.

All ASP.NET Core applications have dependencies on a number of different external libraries, which are managed through the NuGet package manager. These dependencies are listed in the project, but the files of the libraries themselves aren't included. Before you can build and run your application, you need to ensure there are local copies of each dependency in your project folder. The first command, `dotnet restore`, ensures your application's NuGet dependencies are copied to your project folder.

ASP.NET Core projects list their dependencies in the project's csproj file. This is an XML file that lists each dependency as a `PackageReference` node. When you run `dotnet restore`, it uses this file to establish which NuGet packages to download and copy to your project folder. Any dependencies listed are available for use in your application.

You can compile your application using `dotnet build`. This will check for any errors in your application and, if there are no issues, will produce an output that can be run using `dotnet run`.

Each command contains a number of switches that can modify its behavior. To see the full list of available commands, run

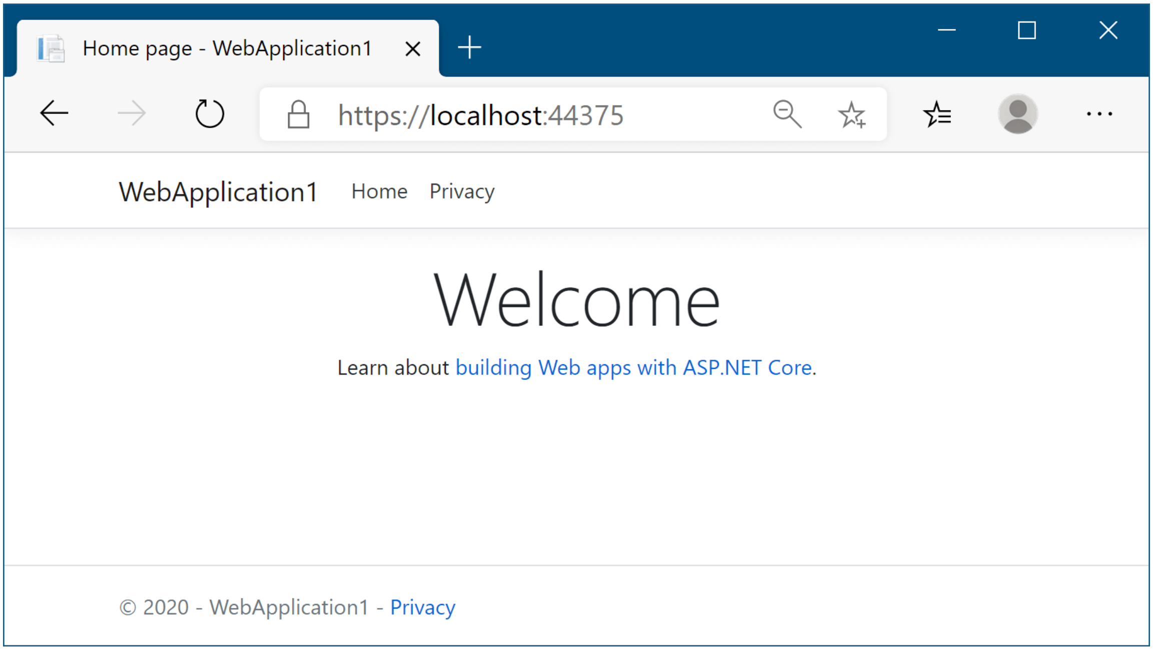
```
dotnet –help
```

or to see the options available for a particular command, `new` for example, run

```
dotnet new --help
```

2.3 Running the web application

You're ready to run your first application and there are a number of different ways to go about it. In Visual Studio, you can either click the green arrow on the toolbar next to IIS Express, or press the F5 shortcut. Visual Studio will automatically open a web browser window for you with the appropriate URL and, after a second or two, you should be presented with your brand-new application, as shown in figure 2.6. Alternatively, you can run the application from the command line with the .NET CLI tools using `dotnet run` and open the URL in a web browser manually, using the address provided on the command line.



Welcome

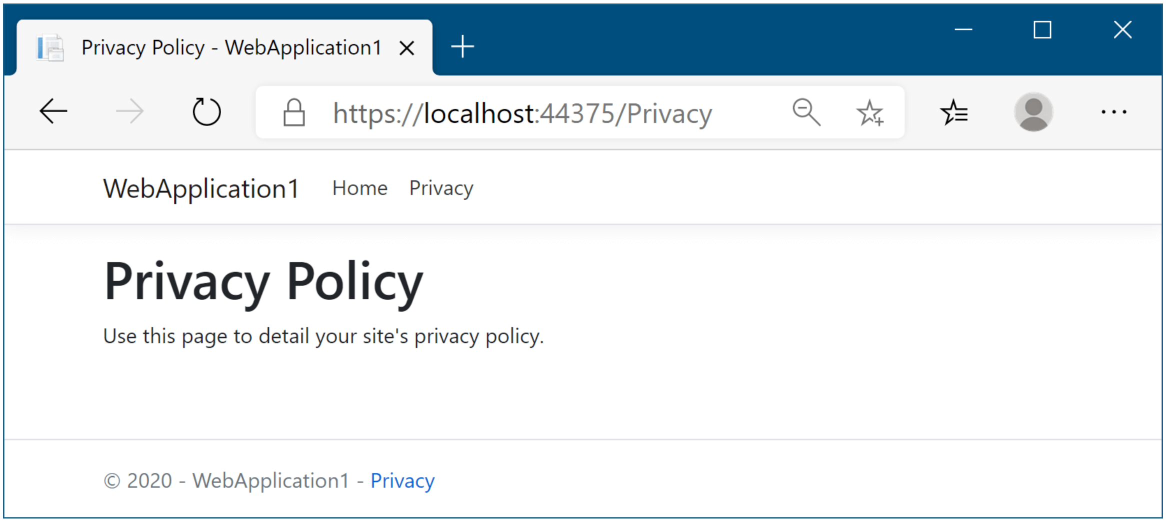
Learn about building Web apps with ASP.NET Core.

© 2020 - WebApplication1 - Privacy

Figure 2.6 The homepage of your new ASP.NET Core application. When you run from Visual Studio, by default IIS Express chooses a random port. If you're running from the command line with `dotnet run`, your application will be available at http://localhost:5000 and https://localhost:5001.

> **TIP** The first time you run the application from Visual Studio you will be prompted to install the development certificate. Doing so ensures your browser doesn't display warnings about an invalid certificate. See chapter 18 for more about HTTPS certificates.

By default, this page shows a simple Welcome banner, and a link to the official Microsoft documentation for ASP.NET Core. At the top of the page are two links: Home and Privacy. The Home link is the page you're currently on. Clicking Privacy will take you to a new page, as shown in figure 2.7. As you'll see shortly, you'll use Razor Pages in your application to define these two pages and to build the HTML they display.


Privacy Policy - WebApplication1
https://localhost:44375/Privacy
WebApplication1
Home
Privacy

Privacy Policy

Use this page to detail your site's privacy policy.

© 2020 - WebApplication1 - Privacy

Figure 2.7 The Privacy page of your application. You can navigate between the two pages of the application using the Home and Privacy links in the application’s header. The app generates the content of the pages using Razor Pages.

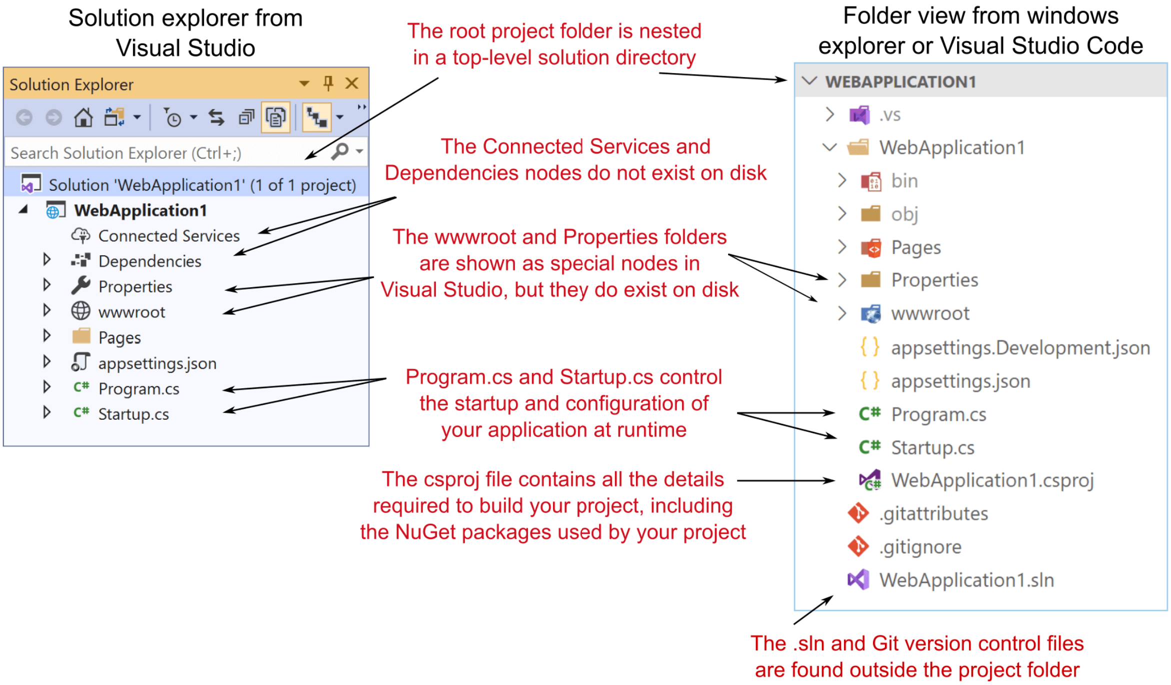
At this point, you need to notice a couple of things. First, the header containing the links and the application title “WebApplication1” is the same on both pages. Second, the title of the page, as shown in the tab of the browser, changes to match the current page. You’ll see how to achieve these features in chapter 7, when we discuss the rendering of HTML using Razor templates.

> **NOTE** You can only view the application on the same computer that is running it at the moment, your application isn’t exposed to the internet yet. You’ll learn how to publish and deploy your application in chapter 16.

There isn’t any more to the user experience of the application at this stage. Click around a little and, once you’re happy with the behavior of the application, roll up your sleeves—it’s time to look at some code!

2.4 Understanding the project layout

When you’re new to a framework, creating an application from a template like this can be a mixed blessing. On the one hand, you can get an application up and running quickly, with little input required on your part. Conversely, the number of files can sometimes be overwhelming, leaving you scratching your head working out where to start. The basic web application template doesn’t contain a huge number of files and folders, as shown in figure 2.8, but I’ll run through the major ones to get you oriented.



Solution explorer from Visual Studio

Folder view from windows explorer or Visual Studio Code

The root project folder is nested in a top-level solution directory

The Connected Services and Dependencies nodes do not exist on disk

The wwwroot and Properties folders are shown as special nodes in Visual Studio, but they do exist on disk

Program.cs and Startup.cs control the startup and configuration of your application at runtime

The csproj file contains all the details required to build your project, including the NuGet packages used by your project

The .sln and Git version control files are found outside the project folder

Figure 2.8 The Solution Explorer and folder on disk for a new ASP.NET Core application. The Solution Explorer also displays the Connected Services and Dependencies nodes, which list NuGet and other dependencies, though the folders themselves don't exist on disk.

The first thing to notice is that the main project, WebApplication1, is nested in a top-level directory with the name of the solution, also WebApplication1 in this case. Within this top-level folder, you'll also find the solution (.sln) file for use by Visual Studio and files related to Git version control,[13] though these are hidden in Visual Studio's Solution Explorer view.

> **NOTE** Visual Studio uses the concept of a solution to work with multiple projects. The example solution only consists of a single project, which is listed in the .sln file. If you use a CLI template to create your project, you won't have a .sln or Git files unless you generate them explicitly using other .NET CLI templates.

Inside the solution folder, you'll find your project folder, which in turn contains three subfolders—Pages, Properties, and wwwroot. Pages (unsurprisingly) contains the Razor Pages

[13] The Git files will only be added if you choose Add to Source Control in Visual Studio. You don't have to use Git, but I strongly recommend using some sort of version control when you build applications. If you're somewhat familiar with Git, but still find it a bit daunting, and a rebase terrifying, I highly recommend reading http://think-like-a-git.net/. It helped me achieve Git enlightenment.

files you'll use to build your application. The Properties folder contains a single file, launchSettings.json, which controls how Visual Studio will run and debug the application. The wwwroot folder is special, in that it's the only folder in your application that browsers are allowed to directly access when browsing your web app. You can store your CSS, JavaScript, images, or static HTML files in here and browsers will be able to access them. They won't be able to access any file that lives outside of wwwroot.

Although the wwwroot and Properties folders exist on disk, you can see that Solution Explorer shows them as special nodes, out of alphabetical order, near the top of your project. You've got two more special nodes in the project, Dependencies and Connected Services, but they don't have a corresponding folder on disk. Instead, they show a collection of all the dependencies, such as NuGet packages and remote services that the project relies on.

In the root of your project folder, you'll find two JSON files: appsettings.json and appsettings.Development.json. These provide configuration settings which are used at runtime to control the behavior of your app.

The most important file in your project is WebApplication1.csproj, as it describes how to build your project. Visual Studio doesn't explicitly show the csproj file, but you can edit it if you double-click the project name in Solution Explorer. We'll have a closer look at this file in the next section.

Finally, Visual Studio shows two C# files in the project folder—Program.cs and Startup.cs. In sections 2.6 and 2.7, you'll see how these fundamental classes are responsible for configuring and running your application.

2.5 The csproj project file: defining your dependencies

The csproj file is the project file for .NET applications and contains the details required for the .NET tooling to build your project. It defines the type of project being built (web app, console app, or library), which platform the project targets (.NET Core 3.1, .NET 5, and so on), and which NuGet packages the project depends on.

The project file has been a mainstay of .NET applications, but in ASP.NET Core it has had a facelift to make it easier to read and edit. These changes include:

- *No GUIDs*—Previously, Global Unique Identifiers (GUIDs) were used for many things, now they're rarely used in the project file.
- *Implicit file includes*—Previously, every file in the project had to be listed in the csproj file for it to be included in the build. Now, files are automatically compiled.
- *No paths to NuGet package dlls*—Previously, you had to include the path to the dll files contained in NuGet packages in the csproj, as well as listing the dependencies in a packages.xml file. Now, you can reference the NuGet package directly in your csproj, and don't need to specify the path on disk.

All of these changes combine to make the project file far more compact than you'll be used to from previous .NET projects. The following listing shows the entire csproj file for your sample app.

Listing 2.1 The csproj project file, showing SDK, target framework, and references

```
<Project Sdk="Microsoft.NET.Sdk.Web">                                  #A
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>                   #B
  </PropertyGroup>
</Project>
```

#A The SDK attribute specifies the type of project you're building.
#B The TargetFramework is the framework you'll run on, in this case, .NET Core 3.1.

For simple applications, you probably won't need to change the project file much. The `Sdk` attribute on the `Project` element includes default settings that describe how to build your project, whereas the `TargetFramework` element describes the framework your application will run on. For .NET Core 3.1 projects, this will have the `netcoreapp3.1` value; if you're running on .NET 5, this would be `net5.0`.

> **TIP** With the new csproj style, Visual Studio users can double-click a project in Solution Explorer to edit the .csproj, without having to close the project first.

The most common changes you'll make to the project file are to add additional NuGet packages using the `PackageReference` element. By default, your app doesn't reference any NuGet packages at all.

Using NuGet libraries in your project

Even though all apps are unique in some way, they also all share common requirements. For example, most apps need to access a database, or manipulate JSON or XML formatted data. Rather than having to reinvent that code in every project, you should use reusable libraries that already exist.

NuGet is the library package manager for .NET, where libraries are packaged into *NuGet packages* and published to https://nuget.org. You can use these packages in your project by referencing the unique package name in your csproj file. These make the package's namespace and classes available in your code files.

You can add a NuGet reference to your project by running `dotnet add package <packagename>` from inside the project folder. This updates your project file with a `<PackageReference>` node and restores the NuGet package for your project. For example, to install the popular Newtonsoft.Json library you would run

```
dotnet add package Newtonsoft.Json
```

This adds a reference to the latest version of the library to your project file, as shown below, and makes the Newtonsoft.Json namespace available in your source code files

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="NewtonSoft.Json" Version="12.0.3" />
  </ItemGroup>
</Project>
```

As a point of interest, there's no officially agreed upon pronunciation for NuGet. Feel free to use the popular "noo-get" or "nugget" styles, or if you're feeling especially posh, "noo-jay"!

The simplified project file format is much easier to edit by hand than previous versions, which is great if you're developing cross-platform. But if you're using Visual Studio, don't feel like you have to take this route. You can still use the GUI to add project references, exclude files, manage NuGet packages, and so on. Visual Studio will update the project file itself, as it always has.

TIP For further details on the changes to the csproj format, see the documentation at https://docs.microsoft.com/en-us/dotnet/core/tools/csproj.

The project file defines everything Visual Studio and the .NET CLI need to build your app. Everything, that is, except the code! In the next section, we'll take a look at the entry point for your ASP.NET Core application—the Program.cs class.

2.6 The Program class: building a web host

All ASP.NET Core applications start in the same way as .NET Console applications—with a Program.cs file. This file contains a `static void Main` function, which is a standard characteristic of console apps. This method must exist and is called whenever you start your web application. In ASP.NET Core applications, it's used to build and run an `IHost` instance, as shown in the following listing, which shows the default Program.cs file. The `IHost` is the core of your ASP.NET Core application, containing the application configuration and the Kestrel server that listens for requests and sends responses.

Listing 2.2 The default Program.cs configures and runs an `IWebHost`

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args)                          #A
            .Build()                                     #B
            .Run();                                      #C
}
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)                  #D
            .ConfigureWebHostDefaults(webBuilder =>      #E
            {
                webBuilder.UseStartup<Startup>();        #F
            };
    }
}
```

#A Create an IHostBuilder using the CreateHostBuilder method .
#B Build and return an instance of IHost from the IHostBuilder.
#C Run the IHost, start listening for requests and generating responses.
#D Create an IHostBuilder using the default configuration.
#E Configure the application to use Kestrel and listen to HTTP requests
#F The Startup class defines most of your application's configuration.

The `Main` function contains all the basic initialization code required to create a web server and to start listening for requests. It uses a `IHostBuilder`, created by the call to `CreateDefaultBuilder`, to define how the `IHost` is configured, before instantiating the `IWebHost` with a call to `Build()`.

> **NOTE** You'll find this pattern of using a builder object to configure a complex object repeated throughout the ASP.NET Core framework. It's a useful technique for allowing users to configure an object, delaying its creation until all configuration has finished. It's one of the patterns described in the "Gang of Four" book, ***Design Patterns: Elements of Reusable Object-Oriented Software*** **(Addison Wesley, 1994).**

Much of your app's configuration takes place in the `IHostBuilder` created by the call to `CreateDefaultBuilder`, but it delegates some responsibility to a separate class, `Startup`. The `Startup` class referenced in the generic `UseStartup<>` method is where you configure your app's services and define your middleware pipeline. In section 2.7, we'll spend a while delving into this crucial class.

At this point, you may be wondering why you need two classes for configuration: `Program` and `Startup`. Why not include all of your app's configuration in one class or the other?

Figure 2.9 shows the typical split of configuration components between `Program` and `Startup`. Generally speaking, `Program` is where you configure the infrastructure of your application, such as the HTTP server, integration with IIS, and configuration sources. In contrast, `Startup` is where you define which components and features your application uses, and the middleware pipeline for your app.

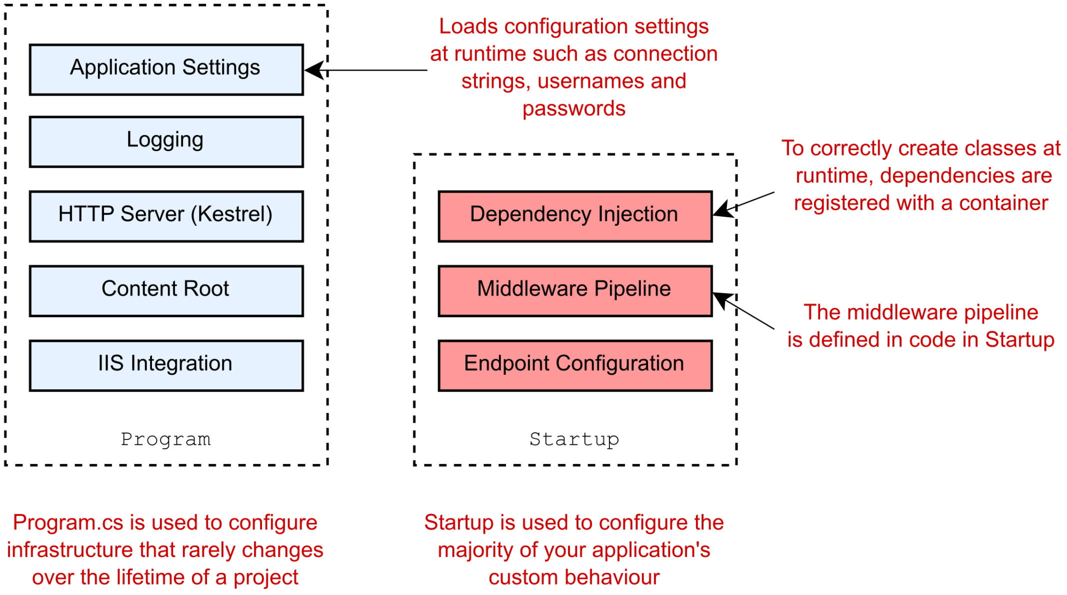

Application Settings
Logging
HTTP Server (Kestrel)
Content Root
IIS Integration
Program
Loads configuration settings at runtime such as connection strings, usernames and passwords
Dependency Injection
Middleware Pipeline
Endpoint Configuration
Startup
To correctly create classes at runtime, dependencies are registered with a container
The middleware pipeline is defined in code in Startup
Program.cs is used to configure infrastructure that rarely changes over the lifetime of a project
Startup is used to configure the majority of your application's custom behaviour

Figure 2.9 The difference in configuration scope for `Program` and `Startup`. `Program` is concerned with infrastructure configuration that will typically remain stable throughout the lifetime of the project. In contrast, you'll often modify `Startup` to add new features and to update application behavior.

The `Program` class for two different ASP.NET Core applications will generally be similar, but the `Startup` classes will often differ significantly (though they generally follow a similar pattern, as you'll see shortly). You'll rarely find that you need to modify `Program` as your application grows, whereas you'll normally update `Startup` whenever you add additional features. For example, if you add a new NuGet dependency to your project, you'll normally need to update `Startup` to make use of it.

The `Program` class is where a lot of app configuration takes place, but in the default templates this is hidden inside the `CreateDefaultBuilder` method. The `CreateDefaultBuilder` method is a static helper method that simplifies the bootstrapping of your app by creating an `IHostBuilder` with some common configuration. In chapter 11, we'll peek inside this method and explore the configuration system, but for now, it's enough to keep figure 2.9 in mind, and to be aware that you can completely change the `IHost` configuration if you need to.

The other helper method used by default is `ConfigureWebHostDefaults`. This uses a `WebHostBuilder` object to configure Kestrel to listen for HTTP requests.

Creating services with the generic host

It might seem strange that you must call `ConfigureWebHostDefaults` as well as `CreateDefaultBuilder`—couldn't we just have one method? Isn't handling HTTP requests the whole *point* of ASP.NET Core?

Well, yes and no! ASP.NET Core 3.0 introduced the concept of a *generic host*. This allows you to use much of the same framework as ASP.NET Core applications to write non-HTTP applications. These apps can be run as console apps or can be installed as Windows services (or as systemd daemons on Linux), to run background tasks or read from message queues, for example.

Kestrel and the web framework of ASP.NET Core builds *on top* of the generic host functionality introduced in ASP.NET Core 3.0. To configure a typical ASP.NET Core app, you configure the generic host features that are common across all apps; features such as configuration, logging, and dependency services. For web applications, you then also configure the services such as Kestrel that are necessary to handle web requests.

In chapter 20 you'll see how to build applications using the generic host for running scheduled tasks and building services.

Once the configuration of the `IHostBuilder` is complete, the call to `Build` produces the `IHost` instance, but the application still isn't handling HTTP requests yet. It's the call to `Run` that starts the HTTP server listening. At this point, your application is fully operational and can respond to its first request from a remote browser.

2.7 The Startup class: configuring your application

As you've seen, `Program` is responsible for configuring a lot of the infrastructure for your app, but you configure some of your app's behavior in `Startup`. The `Startup` class is responsible for configuring two main aspects of your application:

- *Service registration*—Any classes that your application depends on for providing functionality—both those used by the framework and those specific to your application—must be registered so that they can be correctly instantiated at runtime.
- *Middleware and endpoints*—How your application handles and responds to requests.

You configure each of these aspects in its own method in `Startup`, service registration in `ConfigureServices`, and middleware configuration in `Configure`. A typical outline of `Startup` is shown in the following listing.

Listing 2.3 An outline of Startup.cs showing how each aspect is configured

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)    #A
    {
        // method details
    }
    public void Configure(IApplicationBuilder app)               #B
    {
        // method details
    }
}
```

#A Configure services by registering services with the IServiceCollection.
#B Configure the middleware pipeline for handling HTTP requests.

The IHostBuilder created in Program calls ConfigureServices and then Configure, as shown in figure 2.10. Each call configures a different part of your application, making it available for subsequent method calls. Any services registered in the ConfigureServices method are available to the Configure method. Once configuration is complete, an IHost is created by calling Build() on the IHostBuilder.

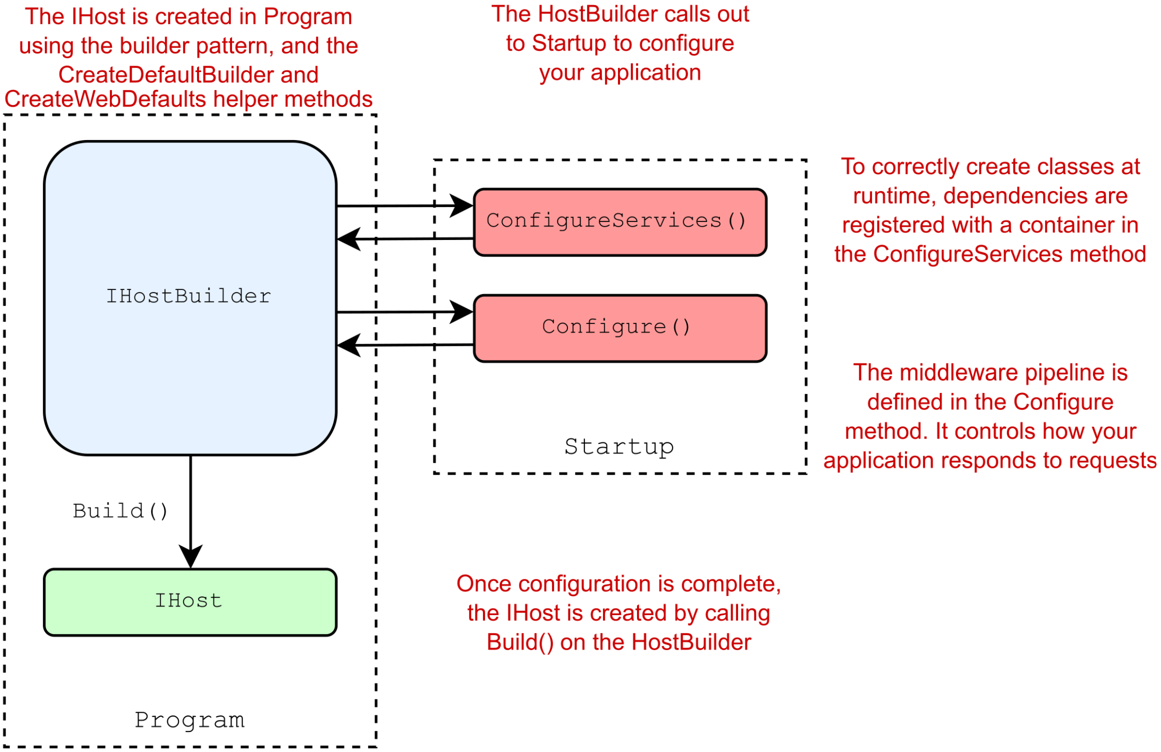

The IHost is created in Program using the builder pattern, and the CreateDefaultBuilder and CreateWebDefaults helper methods
The HostBuilder calls out to Startup to configure your application
To correctly create classes at runtime, dependencies are registered with a container in the ConfigureServices method
The middleware pipeline is defined in the Configure method. It controls how your application responds to requests
Once configuration is complete, the IHost is created by calling Build() on the HostBuilder
IHostBuilder
ConfigureServices()
Configure()
Startup
Build()
IHost
Program

Figure 2.10 The IHostBuilder is created in Program.cs and calls methods on Startup to configure the application's services and middleware pipeline. Once configuration is complete, the IHost is created by calling Build() on the IHostBuilder.

An interesting point about the Startup class is that it doesn't implement an interface as such. Instead, the methods are invoked by using *reflection* to find methods with the predefined names of Configure and ConfigureServices. This makes the class more flexible and enables you to modify the signature of the method to accept additional parameters that are fulfilled automatically. I'll cover how this works in detail in chapter 10, for now it's enough to know that anything that's configured in ConfigureServices can be accessed by the Configure method.

DEFINITION *Reflection* in .NET allows you to obtain information about types in your application at runtime. You can use reflection to create instances of classes at runtime, and to invoke and access them.

Given how fundamental the `Startup` class is to ASP.NET Core applications, the rest of section 2.7 walks you through both `ConfigureServices` and `Configure`, to give you a taste of how they're used. I won't explain them in detail (we have the rest of the book for that!), but you should keep in mind how they follow on from each other and how they contribute to the application configuration as a whole.

2.7.1 Adding and configuring services

ASP.NET Core uses small, modular components for each distinct feature. This allows individual features to evolve separately, with only a loose coupling to others, and is generally considered good design practice. The downside to this approach is that it places the burden on the consumer of a feature to correctly instantiate it. Within your application, these modular components are exposed as one or more *services* that are used by the application.

DEFINITION Within the context of ASP.Net Core, *service* refers to any class that provides functionality to an application and could be classes exposed by a library or code you've written for your application.

For example, in an e-commerce app, you might have a `TaxCalculator` that calculates the tax due on a particular product, taking into account the user's location in the world. Or you might have a `ShippingCostService` that calculates the cost of shipping to a user's location. A third service, `OrderTotalCalculatorService`, might use both of these services to work out the total price the user must pay for an order. Each service provides a small piece of independent functionality, but you can combine them to create a complete application. This is known as the *single responsibility principle*.

DEFINITION The *single responsibility principle* (SRP) states that every class should be responsible for only a single piece of functionality—it should only need to change if that required functionality changes. It's one of the five main design principles promoted by Robert C. Martin in *Agile Software Development, Principles, Patterns, and Practices* (Pearson, 2011).

The `OrderTotalCalculatorService` needs access to an instance of `ShippingCostService` and `TaxCalculator`. A naïve approach to this problem is to use the `new` keyword and create an instance of a service whenever you need it. Unfortunately, this tightly couples your code to the specific implementation you're using and can completely undo all the good work achieved by modularizing the features in the first place. In some cases, it may break the SRP by making you perform initialization code in addition to using the service you created.

One solution to this problem is to make it somebody else's problem. When writing a service, you can declare your dependencies and let another class fill those dependencies for you. Your service can then focus on the functionality for which it was designed, instead of trying to work out how to build its dependencies.

This technique is called dependency injection or the inversion of control (IoC) principle and is a well-recognized *design pattern* that is used extensively.

DEFINITION *Design patterns* are solutions to common software design problems.

Typically, you'll register the dependencies of your application into a "container," which can then be used to create any service. This is true for both your own custom application services and the framework services used by ASP.NET Core. You must register each service with the container before it can be used in your application.

NOTE I'll describe the dependency inversion principle and the IoC container used in ASP.NET Core in detail in chapter 10.

In an ASP.NET Core application, this registration is performed in the `ConfigureServices` method. Whenever you use a new ASP.NET Core feature in your application, you'll need to come back to this method and add in the necessary services. This is not as arduous as it sounds, as shown here, taken from the example application.

Listing 2.4 `Startup.ConfigureServices`**: adding services to the IoC container**

```
public class Startup
{
    // This method gets called by the runtime.
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }
}
```

You may be surprised that a complete Razor Pages application only includes a single call to add the necessary services, but the `AddRazorPages()` method is an extension method that encapsulates all the code required to set up the Razor Pages services. Behind the scenes, it adds various Razor services for rendering HTML, formatter services, routing services, and many more!

As well as registering framework-related services, this method is where you'd register any custom services you have in your application, such as the example `TaxCalculator` discussed previously. The `IServiceCollection` is a list of every known service that your application will need to use. By adding a new service to it, you ensure that whenever a class declares a dependency on your service, the IoC container knows how to provide it.

With your services all configured, it's time to move on to the final configuration step, defining how your application responds to HTTP requests.

2.7.2 Defining how requests are handled with middleware

So far, in the `IHostBuilder` and `Startup` class, you've defined the infrastructure of the application and registered your services with the IoC container. In the final configuration method of `Startup`, `Configure`, you define the middleware pipeline for the application, which is what defines how your app handles HTTP requests. Here's the `Configure` method for the template application.

Listing 2.5 `Startup.Configure`: defining the middleware pipeline

```
public class Startup
{
    public void Configure(
        IApplicationBuilder app,                                #A
        IWebHostEnvironment env)                                #B
    {
        if (env.IsDevelopment())                                #C
        {
            app.UseDeveloperExceptionPage();                    #D
        }
        else
        {
            app.UseExceptionHandler("/Error");                 #E
            app.UseHsts();                                     #E
        }

        app.UseHttpsRedirection();

        app.UseStaticFiles();                                  #F

        app.UseRouting();                                      #G
        app.UseAuthorization();                                #H

        app.UseEndpoints(endpoints =>                          #H
        {
            endpoints.MapRazorPages();
        }
    }
}
```

#A The IApplicationBuilder is used to build the middleware pipeline.
#B Other services can be accepted as parameters.
#C Different behavior when in development or production
#D Only runs in a development environment
#E Only runs in a production environment
#F Adds the static file middleware
#G Adds the endpoint routing middleware, which determines which endpoint to execute
#H Adds the endpoint middleware, which executes a Razor Page to generate an HTML response

As I described previously, middleware consists of small components that execute in sequence when the application receives an HTTP request. They can perform a whole host of functions, such as logging, identifying the current user for a request, serving static files, and handling errors.

The IApplicationBuilder that's passed to the Configure method is used to define the order in which middleware executes. The order of the calls in this method is important, as the order in which they're added to the builder is the order they'll execute in the final pipeline. Middleware can only use objects created by previous middleware in the pipeline—it can't access objects created by later middleware.

WARNING It's important that you consider the order of middleware when adding it to the pipeline. Middleware can only use objects created by earlier middleware in the pipeline.

You should also note that an IWebHostEnvironment parameter is used to provide different behavior when you're in a development environment. When you're running in development (when EnvironmentName is set to "Development"), the Configure method adds one piece of exception-handling middleware to the pipeline; in production, it adds a different one.

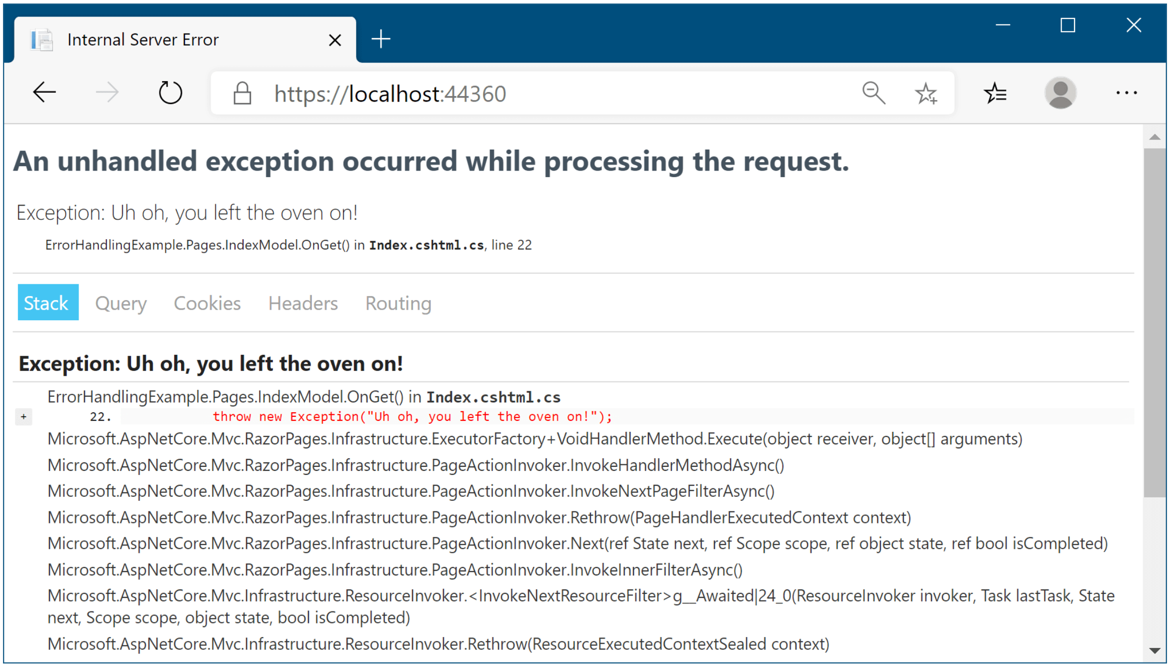
The IWebHostEnvironment object contains details about the current environment, as determined by the IHostBuilder in Program. It exposes a number of properties:

- ContentRootPath—Location of the working directory for the app, typically the folder in which the application is running
- WebRootPath—Location of the wwwroot folder that contains static files
- EnvironmentName—Whether the current environment is a development or production environment

IWebHostEnvironment is already set by the time Startup is invoked; you can't change these values using the application settings in Startup. EnvironmentName is typically set externally by using an environment variable when your application starts.

NOTE You'll learn about hosting environments and how to change the current environment in chapter 11.

In development, DeveloperExceptionPageMiddleware (added by the UseDeveloperExceptionPage() call) ensures that, if your application throws an exception that isn't caught, you'll be presented with as much information as possible in the browser to diagnose the problem, as shown in figure 2.11. It's akin to the "yellow screen of death" in the previous version of ASP.NET, but this time it's white, not yellow!



An unhandled exception occurred while processing the request.

Exception: Uh oh, you left the oven on!

ErrorHandlingExample.Pages.IndexModel.OnGet() in **Index.cshtml.cs**, line 22

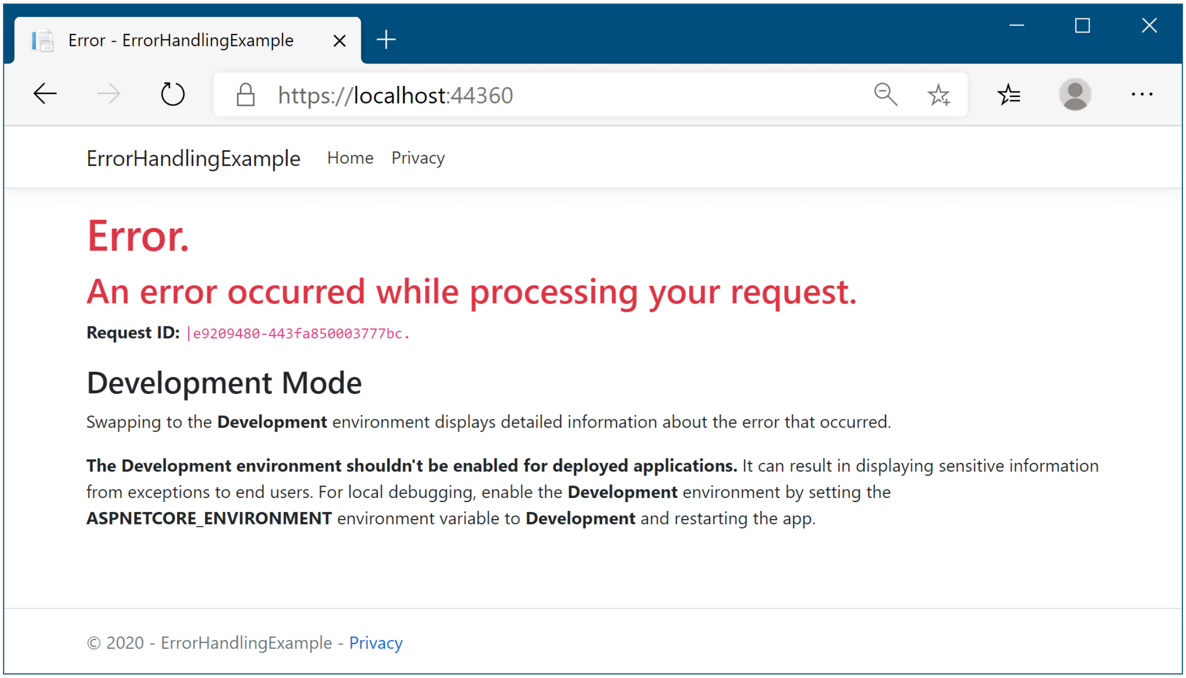
Exception: Uh oh, you left the oven on!

```
ErrorHandlingExample.Pages.IndexModel.OnGet() in Index.cshtml.cs
+    22.              throw new Exception("Uh oh, you left the oven on!");
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.ExecutorFactory+VoidHandlerMethod.Execute(object receiver, object[] arguments)
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker.InvokeHandlerMethodAsync()
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker.InvokeNextPageFilterAsync()
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker.Rethrow(PageHandlerExecutedContext context)
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker.Next(ref State next, ref Scope scope, ref object state, ref bool isCompleted)
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker.InvokeInnerFilterAsync()
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeNextResourceFilter>g__Awaited|24_0(ResourceInvoker invoker, Task lastTask, State next, Scope scope, object state, bool isCompleted)
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Rethrow(ResourceExecutedContextSealed context)
```

Figure 2.11 The developer exception page contains many different sources of information to help you diagnose a problem, including the exception stack trace and details about the request that generated the exception.

NOTE The default templates also add `HstsMiddleware` in production which sets security headers in your response, in line with industry best practices. See chapter 18 for details about this and other security-related middleware.

When you're running in a production environment, exposing this amount of data to users would be a big security risk. Instead, `ExceptionHandlerMiddleware` is registered so that, if users encounter an exception in your method, they will be presented with a friendly error page that doesn't reveal the source of the problems. If you run the default template in production mode and trigger an error, then you'll be presented with the message shown in figure 2.12 instead. Obviously, you'd need to update this page to be more visually appealing and more user-friendly, but at least it doesn't reveal the inner workings of your application!



Error.

An error occurred while processing your request.

Request ID: |e9209480-443fa850003777bc.

Development Mode

Swapping to the **Development** environment displays detailed information about the error that occurred.

The Development environment shouldn't be enabled for deployed applications. It can result in displaying sensitive information from exceptions to end users. For local debugging, enable the **Development** environment by setting the **ASPNETCORE_ENVIRONMENT** environment variable to **Development** and restarting the app.

© 2020 - ErrorHandlingExample - Privacy

Figure 2.12 The default exception-handling page. In contrast to the developer exception page, this doesn't reveal any details about your application to users. In reality, you'd update the message to something more user-friendly.

The next piece of middleware added to the pipeline is the `HttpsRedirectionMiddleware`, using the statement:

```
app.UseHttpsRedirection();
```

This ensures your application only responds to secure (HTTPS) requests and is an industry best practice. We'll look more at HTTPS in chapter 18. The `StaticFileMiddleware` is added to the pipeline next using this statement:

```
app.UseStaticFiles();
```

This middleware is responsible for handling requests for static files such as CSS files, JavaScript files, and images. When a request arrives at the middleware, it checks to see if the request is for an existing file. If it is, then the middleware returns the file. If not, the request is ignored and the next piece of middleware can attempt to handle the request. Figure 2.13 shows how the request is processed when a static file is requested. When the static-file middleware handles a request, other middleware that comes later in the pipeline, such as the routing middleware or the endpoint middleware, won't be called at all.

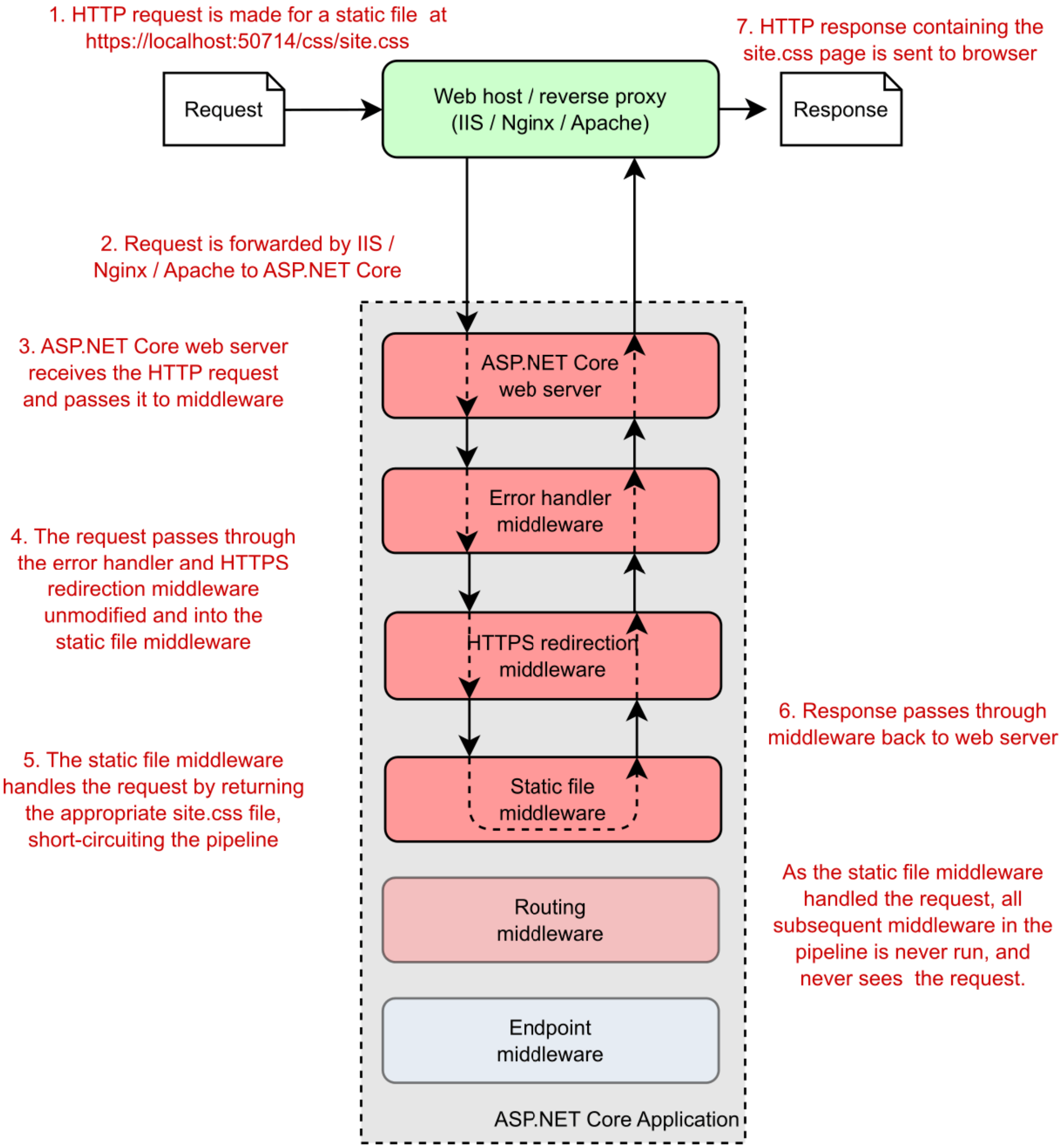

1. HTTP request is made for a static file at https://localhost:50714/css/site.css
Request
Web host / reverse proxy (IIS / Nginx / Apache)
7. HTTP response containing the site.css page is sent to browser
Response
2. Request is forwarded by IIS / Nginx / Apache to ASP.NET Core
3. ASP.NET Core web server receives the HTTP request and passes it to middleware
ASP.NET Core web server
Error handler middleware
4. The request passes through the error handler and HTTPS redirection middleware unmodified and into the static file middleware
HTTPS redirection middleware
6. Response passes through middleware back to web server
5. The static file middleware handles the request by returning the appropriate site.css file, short-circuiting the pipeline
Static file middleware
Routing middleware
As the static file middleware handled the request, all subsequent middleware in the pipeline is never run, and never sees the request.
Endpoint middleware
ASP.NET Core Application

Figure 2.13 An overview of a request for a static file at /css/site.css for an ASP.NET Core application. The request passes through the middleware pipeline until it's handled by the static file middleware. This returns the requested CSS file as the response, which passes back to the web server. The endpoint middleware is never invoked and never sees the request.

Which brings us to the most substantial pieces of middleware in the pipeline: the routing middleware and the endpoint middleware. Together, this pair of middleware are responsible for interpreting the request to determine which Razor Page to invoke, for reading parameters from the request, and for generating the final HTML. Despite that, very little configuration is required—you need only to add the middleware to the pipeline and specify that you wish to use Razor Page endpoints by calling `MapRazorPages`. For each request, the routing middleware uses the request's URL to determine which Razor Page to invoke. The endpoint middleware actually executes the Razor Page to generate the HTML response.

> **NOTE The default templates also add the `AuthorizationMiddleware` *between* the routing middleware and the endpoint middleware. This allows the authorization middleware to decide whether to allow access *before* the Razor Page is executed. You'll learn more about this approach in chapter 5 on routing and chapter 15 on authorization.**

Phew! You've finally finished configuring your application with all the settings, services, and middleware it needs. Configuring your application touches on a wide range of different topics that we'll delve into further throughout the book, so don't worry if you don't fully understand all the steps yet.

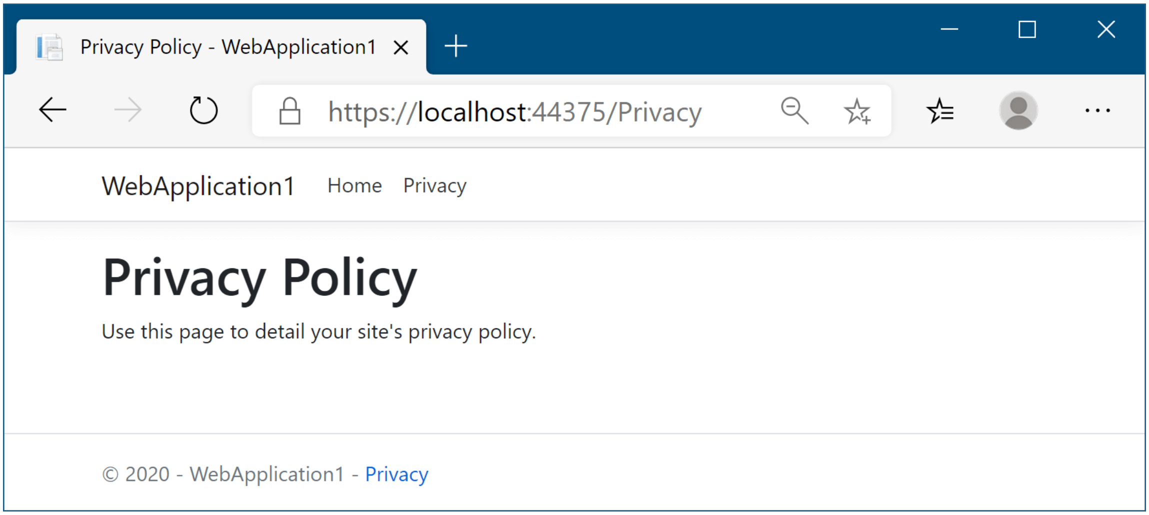
Once the application is configured, it can start handling requests. But *how* does it handle them? I've already touched on `StaticFileMiddleware`, which will serve the image and CSS files to the user, but what about the requests that require an HTML response? In the rest of this chapter, I'll give you a glimpse into Razor Pages and how they generate HTML.

2.8 Generating responses with Razor Pages

When an ASP.NET Core application receives a request, it progresses through the middleware pipeline until a middleware component can handle it, as you saw in figure 2.13. Normally, the final piece of middleware in a pipeline is the endpoint middleware. This middleware works with the routing middleware to match a request URL's path to a configured route, which defines which Razor Page to invoke.

> **DEFINITION A path is the remainder of the request URL, once the domain has been removed. For example, for a request to www.microsoft.com/account/manage, the path is /account/manage.**

Once a Razor Page has been selected, the routing middleware stores a note of the selected Razor Page in the request's `HttpContext` and continues executing the middleware pipeline. Eventually the request will reach the endpoint middleware. The endpoint middleware executes the Razor Page to generate the HTML response, and sends it back to the browser, as shown in figure 2.14.


Privacy Policy - WebApplication1
https://localhost:44375/Privacy
WebApplication1
Home
Privacy

Privacy Policy

Use this page to detail your site's privacy policy.

© 2020 - WebApplication1 - Privacy

Figure 2.14 Rendering a Razor template to HTML. The Razor Page is selected based on the URL page /Privacy and is executed to generate the HTML

In the next section we'll look at how Razor Pages generate HTML using the Razor syntax. After that we'll look at how you can use Page Handlers to add business logic and behavior to your Razor Pages.

2.8.1 Generating HTML with Razor Pages

Razor Pages are stored in cshtml files (a portmanteau of cs and html) within the Pages folder of your project. In general, the routing middleware maps request URL paths to a single Razor Page by looking in the Pages folder of your project for a Razor Page with the same path. For example, you can see in figure 2.14 that the Privacy page of your app corresponds to the path `/Privacy` in the browser's address bar. If you look inside the Pages folder of your project, you'll find the Privacy.cshtml file, shown in the listing below.

Listing 2.6 The Privacy.cshtml Razor Page

```
@page                                                         #A
@model PrivacyModel                                           #B
@{
    ViewData["Title"] = "Privacy Policy";                     #C
}
<h1>@ViewData["Title"]</h1>                                   #D

<p>Use this page to detail your site's privacy policy.</p>   #E
```

#A Indicates that this is a Razor Page
#B Links the Razor Page to a specific PageModel
#C C# code that doesn't write to the response
#D HTML with dynamic C# values written to the response

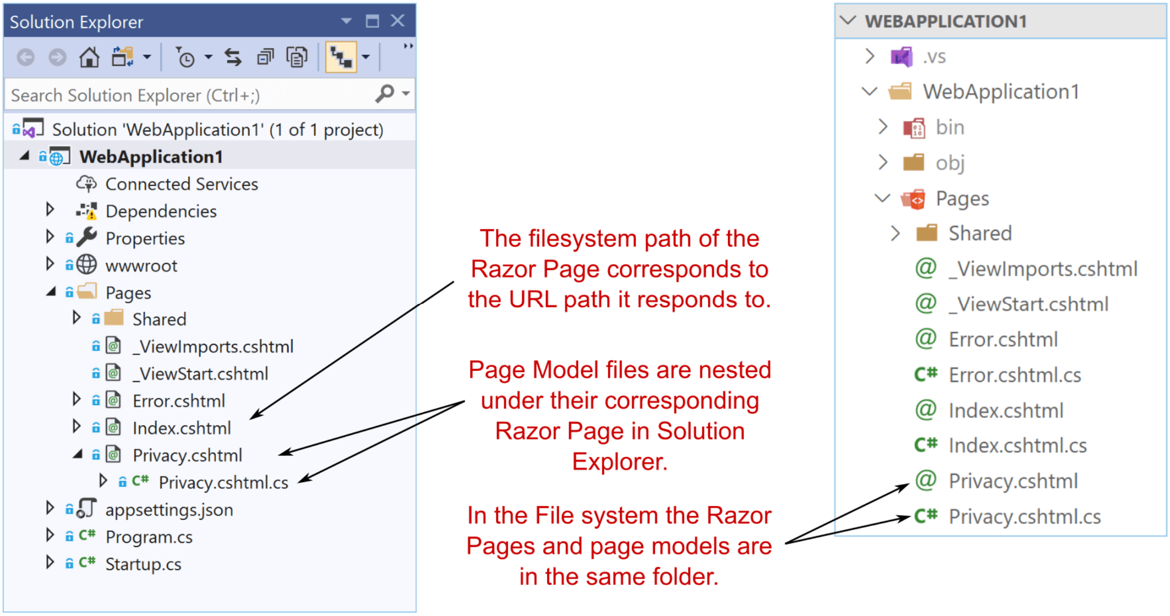
#E Standalone, static HTML

Razor Pages use a templating syntax, called *Razor*, that combines static HTML with dynamic C# code and HTML generation. The `@page` directive on the first line of the Razor Page is the most important. This directive must always be placed on the first line of the file, as it tells ASP.NET Core that the cshtml file is a Razor Page. Without it, you won't be able to view your page correctly.

The next line of the Razor Page defines which `PageModel` in your project the Razor Page is associated with:

```
@model PrivacyModel
```

In this case the `PageModel` is called `PrivacyModel`, and it follows the standard convention for naming Razor Page models. You can find this class in the Privacy.cshtml.cs file in the Pages folder of your project, as shown in figure 2.15. Visual Studio nests these file underneath the Razor Page .cshtml files in Solution explorer. We'll look at the page model in the next section.



The filesystem path of the Razor Page corresponds to the URL path it responds to.

Page Model files are nested under their corresponding Razor Page in Solution Explorer.

In the File system the Razor Pages and page models are in the same folder.

Figure 2.15 By convention page models for Razor Pages are placed in a file with the same name as the Razor Page with a .cs suffix appended. Visual Studio nests these files under the Razor Page in solution explorer.

In addition to the `@page` and `@model` directives, you can see that static HTML is always valid in a Razor Page and will be rendered "as is" in the response.

```
<p>Use this page to detail your site's privacy policy.</p>
```

You can also write ordinary C# code in Razor templates by using this construct

```
@{ /* C# code here */ }
```

Any code between the curly braces will be executed but won't be written to the response. In the listing you're setting the title of the page by writing a key to the `ViewData` dictionary, but you aren't writing anything to the response at this point:

```
@{
    ViewData["Title"] = "Privacy Policy";
}
```

Another feature shown in this template is dynamically writing C# variables to the HTML stream using the `@` symbol. This ability to combine dynamic and static markup is what gives Razor Pages their power. In the example, you're fetching the `"Title"` value from the `ViewData` dictionary and writing the values to the response inside an `<h1>` tag:

```
<h1>@ViewData["Title"]</h1>
```

At this point, you might be a little confused by the template from the listing when compared to the output shown in figure 2.14. The title and the static HTML content appear in both the listing and figure, but some parts of the final web page don't appear in the template. How can that be?

Razor Pages have the concept of *layouts*, which are "base" templates that define the common elements of your application, such as headers and footers. The HTML of the layout combines with the Razor Page template to produce the final HTML that's sent to the browser. This prevents you having to duplicate code for the header and footer in every page and means that, if you need to tweak something, you'll only need to do it in one place.

> **NOTE** **I'll cover Razor templates, including layouts, in detail in chapter 7. You can find layouts in the Pages/Shared folder of your project.**

As you've already seen, you can include C# code in your Razor Pages by using curly braces `@{ }`, but generally speaking you want to limit the code in your cshtml to presentational concerns only. Complex logic, code to access services such as a database, and data manipulation should be handled in the `PageModel` instead.

2.8.2 Handling request logic with PageModels and handlers

As you've already seen, the `@page` directive in a cshtml file marks the page as a Razor Page, but most Razor Pages also have an associated *page model*. By convention, this is placed in a file commonly known as a "code behind" file which has a cs extension, as you saw in figure 2.15. Page models should derive from the `PageModel` base class, and typically contain one or more methods called *page handlers*, that define how to handle requests to the Razor Page.

DEFINITION A *page handler* is a method that runs in response to a request. Razor Page models must be derived from the PageModel class. They can contain multiple page handlers, though typically they only contain one or two.

The listing below shows the page model for the Privacy.cshtml Razor Page, found in the file Privacy.cshtml.cs.

Listing 2.7 The PrivacyModel in Privacy.cshtmlcs—a Razor Page page model

```
public class PrivacyModel: PageModel                        #A
{
    private readonly ILogger<PrivacyModel> _logger;         #B
    public PrivacyModel(ILogger<PrivacyModel> logger)       #B
    {                                                       #B
        _logger = logger;                                   #B
    }                                                       #B

    public void OnGet()                                     #C
    {
    }
}
```

#A Razor Pages must inherit from PageModel.
#B You can use dependency injection to provide services in the constructor.
#C The default page handler is OnGet. Returning void indicates HTML should be generated

This page model is extremely simple, but it demonstrates a couple of important points:

- Page handlers are driven by convention.
- Page models can use dependency injection to interact with other services.

Page handlers are typically named by convention, based on the *HTTP verb* that they respond to. They return either void, indicating that the Razor Page's template should be rendered, or an IActionResult which contains other instructions for generating the response, redirecting the user to a different page, for example.

DEFINITION Every HTTP request includes a *verb* that indicates the "type" of the request. When browsing a website, the default verb is GET, which *fetches* a page from the server so you can view it. The second most common verb is POST which is used to *send* data to the server, for example when completing a form.

The PrivacyModel contains a single handler, OnGet, which indicates it should run in response to GET requests for the page. As the method returns void, executing the handler will execute the associated Razor template for the page, to generate the HTML.

Dependency injection is used to inject an ILogger<PrivacyModel> instance into the constructor of the page model. The service is unused in this example, but it can be used to record useful information to a variety of destinations, such as the console, to a file, or to a remote logging service. You can access additional services in your page model by accepting them as parameters in the constructor—the ASP.NET Core framework will take care of configuring and injecting an instance of any services you request.

> **NOTE** I describe the dependency inversion principle and the IoC container used in ASP.NET Core in detail in chapter 10. Logging is covered in chapter 17.

Clearly, the `PrivacyModel` page model does not do much in this case, and you may be wondering why it's worth having! If all they do is tell the Razor Page to generate HTML, then why do we need page models at all?

The key thing to remember here is that you now have a framework for performing arbitrarily complex functions in response to a request. You could easily update the handler method to load data from the database, send an email, add a product to a basket, or create an invoice—all in response to a simple HTTP request. This extensibility is where a lot of the power in Razor Pages (and the MVC pattern in general) lies.

The other important point is that you've separated the execution of these methods from the generation of the HTML itself. If the logic changes and you need to add behavior to a page handler, you don't need to touch the HTML generation code, so you're less likely to introduce bugs. Conversely, if you need to change the UI slightly, change the color of the title for example, then your handler method logic is safe.

And there you have it, a complete ASP.NET Core Razor Pages application! Before we move on, we'll take one last look at how our application handles a request. Figure 2.16 shows a request to the `/Privacy` path being handled by the sample application. You've seen everything here already, so the process of handling a request should be familiar. It shows how the request passes through the middleware pipeline before being handled by the endpoint middleware. The Privacy.cshtml Razor Page executes the `OnGet` handler and generates the HTML response, which passes back through the middleware to the ASP.NET Core web server, before being sent to the user's browser.

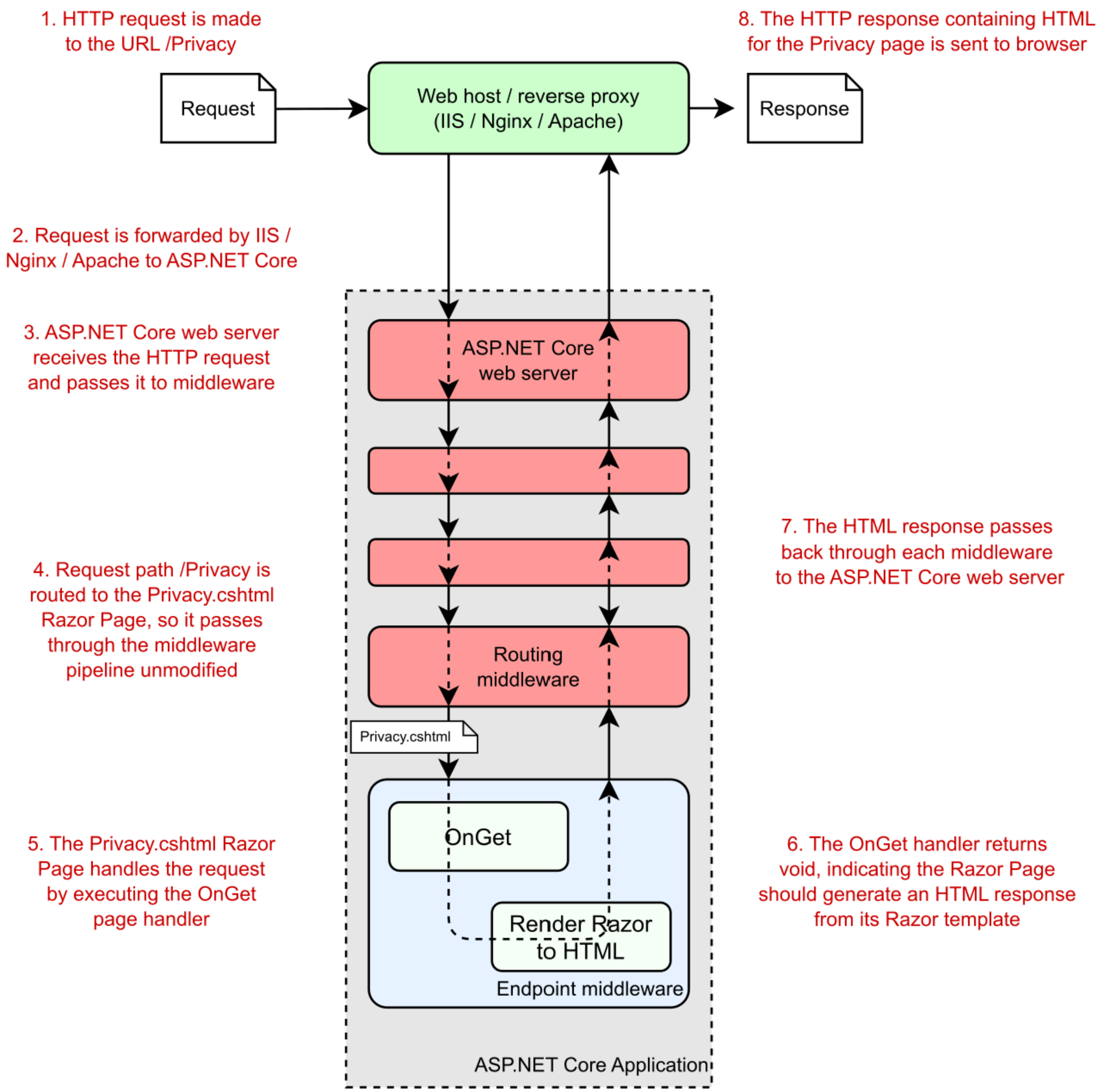

1. HTTP request is made to the URL /Privacy
8. The HTTP response containing HTML for the Privacy page is sent to browser
Request
Web host / reverse proxy (IIS / Nginx / Apache)
Response
2. Request is forwarded by IIS / Nginx / Apache to ASP.NET Core
3. ASP.NET Core web server receives the HTTP request and passes it to middleware
ASP.NET Core web server
7. The HTML response passes back through each middleware to the ASP.NET Core web server
4. Request path /Privacy is routed to the Privacy.cshtml Razor Page, so it passes through the middleware pipeline unmodified
Routing middleware
Privacy.cshtml
5. The Privacy.cshtml Razor Page handles the request by executing the OnGet page handler
OnGet
6. The OnGet handler returns void, indicating the Razor Page should generate an HTML response from its Razor template
Render Razor to HTML
Endpoint middleware
ASP.NET Core Application

Figure 2.16 An overview of a request to the `/Privacy` URL for the sample ASP.NET Razor Pages application. The routing middleware routes the request to `OnGet` handler of the Privacy.cshtml Razor Page. The Razor Page generates an HTML response by executing the Razor template and passes the response back through the middleware pipeline to the browser.

It’s been a pretty intense trip, but you now have a good overview of how an entire application is configured and how it handles a request using Razor Pages. In the next chapter, you’ll take a closer look at the middleware pipeline that exists in all ASP.NET Core applications. You'll learn how it's composed, how you can use it to add functionality to your application, and how you can use it to create simple HTTP services.

2.9 Summary

- The csproj file contains details of how to build your project, including which NuGet packages it depends on. It's used by Visual Studio and the .NET CLI to build your application.
- Restoring the NuGet packages for an ASP.NET Core application downloads all your project's dependencies so it can be built and run.
- Program.cs defines the `static void Main` entry point for your application. This function is run when your app starts, the same as for console applications.
- Program.cs is where you build an `IHost` instance, using an `IHostBuilder`. The helper method, `Host.CreateDefaultBuilder()`creates an `IHostBuilder` that loads configuration settings and sets up logging. Calling `Build()` creates the `IHost` instance.
- The `ConfigureWebHostDefaults` extension method configures the generic host using a `WebHostBuilder`. It configures the Kestrel HTTP server, adds IIS integration if necessary, and specifies the application's `Startup` class.
- You can start the web server and begin accepting HTTP requests by calling `Run` on the `IHost`.
- `Startup` is responsible for service configuration and defining the middleware pipeline.
- All services, both framework and custom application services, must be registered in the call to `ConfigureServices` in order to be accessed later in your application.
- Middleware is added to the application pipeline with `IApplicationBuilder`. Middleware defines how your application responds to requests.
- The order in which middleware is registered defines the final order of the middleware pipeline for the application. Typically, `EndpointMiddleware` is the last middleware in the pipeline. Earlier middleware, such as `StaticFileMiddleware`, will attempt to handle the request first. If the request is handled, `EndpointMiddleware` will never receive the request.
- Razor Pages are located in the Pages folder and are typically named according to the URL path they handle. For example, Privacy.cshtml handles the path `/Privacy`.
- Razor Pages must contain the `@page` directive as the first line of the file.
- Page models derive from the `PageModel` base class and contain page handlers. Page handlers are methods named using conventions that indicate the HTTP verb they handle. For example, `OnGet` handles the `GET` verb.
- Razor templates can contain standalone C#, standalone HTML, and dynamic HTML generated from C# values. By combining all three, you can build highly dynamic applications.

- Razor layouts define common elements of a web page, such as headers and footers. They let you extract this code into a single file, so you don’t have to duplicate it across every Razor template.

3

Handling requests with the middleware pipeline

This chapter covers

- **What middleware is**
- **Serving static files using middleware**
- **Adding functionality using middleware**
- **Combining middleware to form a pipeline**
- **Handling exceptions and errors with middleware**

In the previous chapter, you had a whistle-stop tour of a complete ASP.NET Core application to see how the components come together to create a web application. In this chapter, we focus in on one small subsection: the middleware pipeline.

The middleware pipeline is one of the most important parts of configuration for defining how your application behaves and how it responds to requests. Understanding how to build and compose middleware is key to adding functionality to your applications.

In this chapter, you'll learn what middleware is and how to use it to create a pipeline. You'll see how you can chain multiple middleware components together, with each component adding a discrete piece of functionality. The examples in this chapter are limited to using existing middleware components, showing how to arrange them in the correct way for your application. In chapter 19, you'll learn how to build your own middleware components and incorporate them into the pipeline.

We'll begin by looking at the concept of middleware, all the things you can achieve with it, and how a middleware component often maps to a "cross-cutting concern." These are the functions of an application that cut across multiple different layers. Logging, error handling, and security are classic cross-cutting concerns that are all required by many different parts of your application. As all requests pass through the middleware pipeline, it's the preferred location to configure and handle these aspects.

In section 3.2, I'll explain how you can compose individual middleware components into a pipeline. You'll start out small, with a web app that only displays a holding page. From there, you'll learn how to build a simple static-file server that returns requested files from a folder on disk.

Next, you'll move on to a more complex pipeline containing multiple middleware. You'll use this example to explore the importance of ordering in the middleware pipeline and to see how requests are handled when your pipeline contains more than one middleware.

In section 3.3, you'll learn how you can use middleware to deal with an important aspect of any application: error handling. Errors are a fact of life for all applications, so it's important that you account for them when building your app. As well as ensuring that your application doesn't break when an exception is thrown or an error occurs, it's important that users of your application are informed about what went wrong in a user-friendly way.

You can handle errors in a few different ways, but as one of the classic cross-cutting concerns, middleware is well placed to provide the required functionality. In section 3.3, I'll show how you can use middleware to handle exceptions and errors using existing middleware provided by Microsoft. In particular, you'll learn about three different components:

- `DeveloperExceptionPageMiddleware`—Provides quick error feedback when building an application
- `ExceptionHandlerMiddleware`—Provides a user-friendly generic error page in production
- `StatusCodePagesMiddleware`—Converts raw error status codes into user-friendly error pages

By combining these pieces of middleware, you can ensure that any errors that do occur in your application won't leak security details and won't break your app. On top of that, they will leave users in a better position to move on from the error, giving them as friendly an experience as possible.

You won't see how to build your own middleware in this chapter—instead, you'll see that you can go a long way using the components provided as part of ASP.NET Core. Once you understand the middleware pipeline and its behavior, it will be much easier to understand when and why custom middleware is required. With that in mind, let's dive in!

3.1 What is middleware?

The word *middleware* is used in a variety of contexts in software development and IT, but it's not a particularly descriptive word—what is middleware?

In ASP.NET Core, middleware are C# classes that can handle an HTTP request or response. Middleware can

- Handle an incoming HTTP *request* by generating an HTTP *response*.

- Process an incoming HTTP *request*, modify it, and pass it on to another piece of middleware.
- Process an outgoing HTTP *response*, modify it, and pass it on to either another piece of middleware, or the ASP.NET Core web server.

You can use middleware in a multitude of ways in your own applications. For example, a piece of logging middleware might note when a request arrived and then pass it on to another piece of middleware. Meanwhile, an image-resizing middleware component might spot an incoming request for an image with a specified size, generate the requested image, and send it back to the user without passing it on.

The most important piece of middleware in most ASP.NET Core applications is the `EndpointMiddleware` class. This class normally generates all your HTML pages and API responses and is the focus of most of this book. Like the image-resizing middleware, it typically receives a request, generates a response, and then sends it back to the user, as shown in figure 3.1.

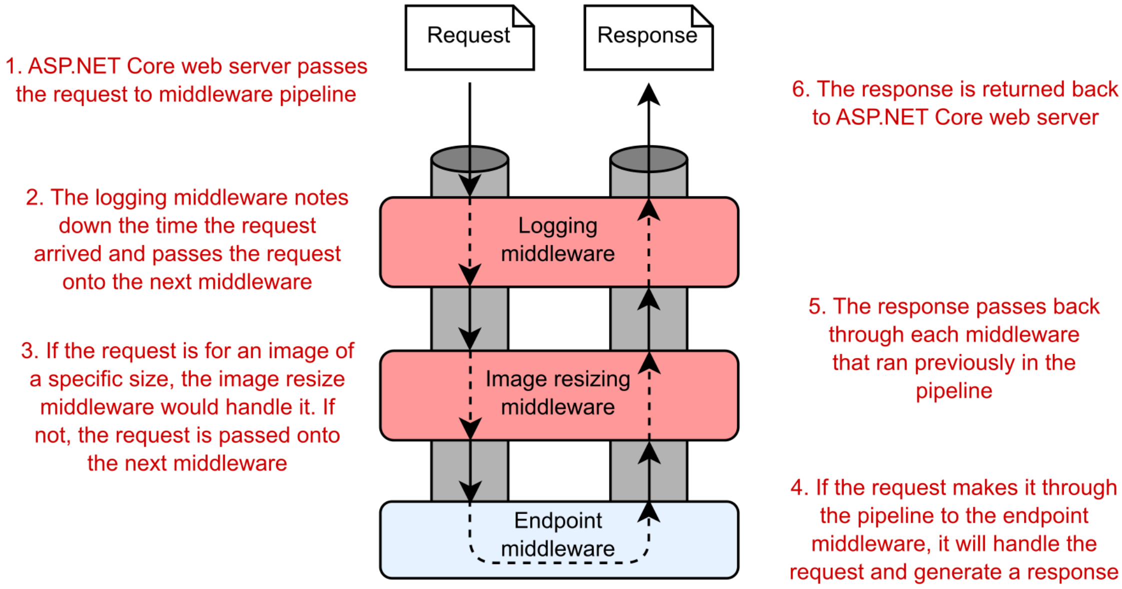

Request
Response
1. ASP.NET Core web server passes the request to middleware pipeline
6. The response is returned back to ASP.NET Core web server
2. The logging middleware notes down the time the request arrived and passes the request onto the next middleware
Logging middleware
5. The response passes back through each middleware that ran previously in the pipeline
3. If the request is for an image of a specific size, the image resize middleware would handle it. If not, the request is passed onto the next middleware
Image resizing middleware
4. If the request makes it through the pipeline to the endpoint middleware, it will handle the request and generate a response
Endpoint middleware

Figure 3.1 Example of a middleware pipeline. Each middleware handles the request and passes it on to the next middleware in the pipeline. After a middleware generates a response, it passes it back through the pipeline. When it reaches the ASP.NET Core web server, the response is sent to the user's browser.

DEFINITION This arrangement, where a piece of middleware can call another piece of middleware, which in turn can call another, and so on, is referred to as a *pipeline*. You can think of each piece of middleware as a section of pipe—when you connect all the sections, a request flows through one piece and into the next.

One of the most common use cases for middleware is for the cross-cutting concerns of your application. These aspects of your application need to occur with every request, regardless of the specific path in the request or the resource requested. These include things like

- Logging each request
- Adding standard security headers to the response
- Associating a request with the relevant user
- Setting the language for the current request

In each of these examples, the middleware would receive a request, modify it, and then pass the request on to the next piece of middleware in the pipeline. Subsequent middleware could use the details added by the earlier middleware to handle the request in some way. For example, in figure 3.2, the authentication middleware associates the request with a user. The authorization middleware uses this detail to verify whether the user has permission to make that specific request to the application or not.

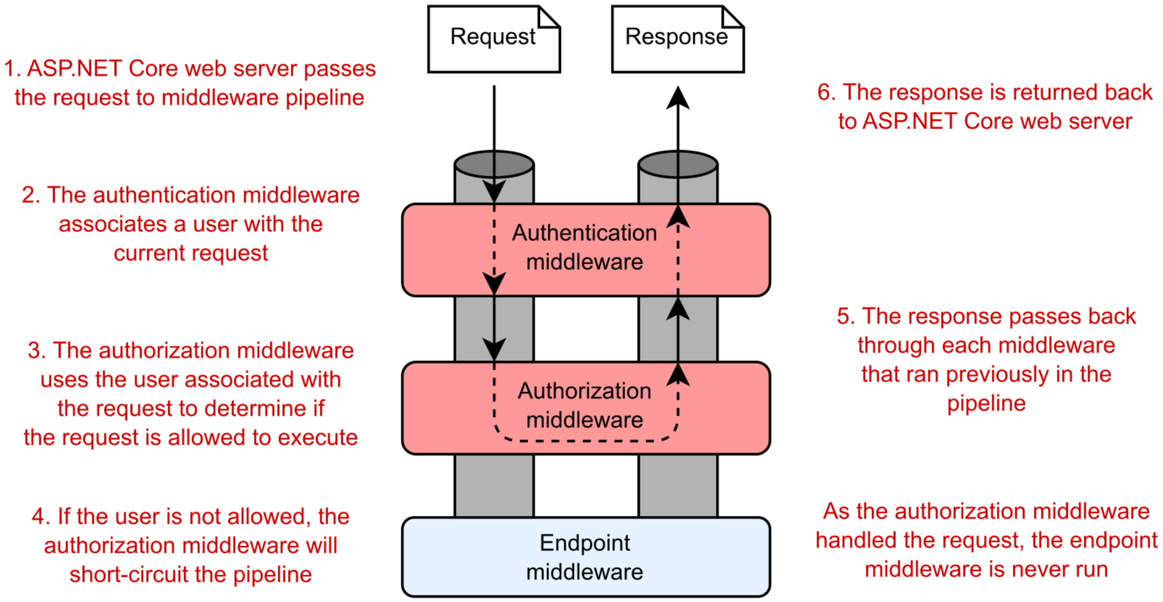

Request
Response
1. ASP.NET Core web server passes the request to middleware pipeline
6. The response is returned back to ASP.NET Core web server
2. The authentication middleware associates a user with the current request
Authentication middleware
5. The response passes back through each middleware that ran previously in the pipeline
3. The authorization middleware uses the user associated with the request to determine if the request is allowed to execute
Authorization middleware
4. If the user is not allowed, the authorization middleware will short-circuit the pipeline
Endpoint middleware
As the authorization middleware handled the request, the endpoint middleware is never run

Figure 3.2 Example of a middleware component modifying the request for use later in the pipeline. Middleware can also short-circuit the pipeline, returning a response before the request reaches later middleware.

If the user has permission, the authorization middleware will pass the request on to the endpoint middleware to allow it to generate a response. If the user doesn't have permission, the authorization middleware can short-circuit the pipeline, generating a response directly. It returns the response to the previous middleware before the endpoint middleware has even seen the request.

A key point to glean from this is that the pipeline is *bidirectional*. The request passes through the pipeline in one direction until a piece of middleware generates a response, at which point the response passes *back* through the pipeline, passing through each piece of middleware for a *second* time, until it gets back to the first piece of middleware. Finally, this first/last piece of middleware will pass the response back to the ASP.NET Core web server.

The HttpContext object

We mentioned the `HttpContext` in chapter 2, and it's sitting behind the scenes here too. The ASP.NET Core web server constructs an `HttpContext`, which the ASP.NET Core application uses as a sort of storage box for a single request. Anything that's specific to this particular request and the subsequent response can be associated with and stored in it. This could include properties of the request, request-specific services, data that's been loaded, or errors that have occurred. The web server fills the initial `HttpContext` with details of the original HTTP request and other configuration details and passes it on to the rest of the application.

All middleware has access to the `HttpContext` for a request. It can use this, for example, to determine whether the request contains any user credentials, which page the request is attempting to access, and to fetch any posted data. It can then use these details to determine how to handle the request.

Once the application has finished processing the request, it will update the `HttpContext` with an appropriate response and return it through the middleware pipeline to the web server. The ASP.NET Core web server then converts the representation into a raw HTTP response and sends it back to the reverse proxy, which forwards it to the user's browser.

As you saw in chapter 2, you define the middleware pipeline in code as part of your initial application configuration in `Startup`. You can tailor the middleware pipeline specifically to your needs—simple apps may need only a short pipeline, whereas large apps with a variety of features may use much more middleware. Middleware is the fundamental source of behavior in your application—ultimately, the middleware pipeline is responsible for responding to any HTTP requests it receives.

Requests are passed to the middleware pipeline as `HttpContext` objects. As you saw in chapter 2, the ASP.NET Core web server builds an `HttpContext` object from an incoming request, which passes up and down the middleware pipeline. When you're using existing middleware to build a pipeline, this is a detail you'll rarely have to deal with. But, as you'll see in the final section of this chapter, its presence behind the scenes provides a route to exerting extra control over your middleware pipeline.

Middleware vs. HTTP modules and HTTP handlers

In the previous version of ASP.NET, the concept of a middleware pipeline isn't used. Instead, you have HTTP modules and HTTP handlers.

An *HTTP handler* is a process that runs in response to a request and generates the response. For example, the ASP.NET page handler runs in response to requests for .aspx pages. Alternatively, you could write a custom handler that returns resized images when an image is requested.

HTTP modules handle the cross-cutting concerns of applications, such as security, logging, or session management. They run in response to the lifecycle events that a request progresses through when it's received by the server. Examples of events include `BeginRequest`, `AcquireRequestState`, and `PostAcquireRequestState`.

This approach works, but it's sometimes tricky to reason about which modules will run at which points. Implementing a module requires a relatively detailed understanding of the state of the request at each individual lifecycle event.

The middleware pipeline makes understanding your application far simpler. The pipeline is completely defined in code, specifying which components should run, and in which order. Behind the scenes, the middleware pipeline in ASP.NET Core is simply a chain of method calls, where each middleware function calls the next in the pipeline.

That's pretty much all there is to the concept of middleware. In the next section, I'll discuss ways you can combine middleware components to create an application, and how to use middleware to separate the concerns of your application.

3.2 Combining middleware in a pipeline

Generally speaking, each middleware component has a single primary concern. It will handle one aspect of a request only. Logging middleware will only deal with logging the request, authentication middleware is only concerned with identifying the current user, and static-file middleware is only concerned with returning static files.

Each of these concerns is highly focused, which makes the components themselves small and easy to reason about. It also gives your app added flexibility; adding a static-file middleware doesn't mean you're forced into having image-resizing behavior or authentication. Each of these features is an additional piece of middleware.

To build a complete application, you compose multiple middleware components together into a pipeline, as shown in the previous section. Each middleware has access to the original request, plus any changes made by previous middleware in the pipeline. Once a response has been generated, each middleware can inspect and/or modify the response as it passes back through the pipeline, before it's sent to the user. This allows you to build complex application behaviors from small, focused components.

In the rest of this section, you'll see how to create a middleware pipeline by composing small components together. Using standard middleware components, you'll learn to create a holding page and to serve static files from a folder on disk. Finally, you'll take another look at the default middleware pipeline you built previously, in chapter 2, and decompose it to understand why it's built like it is.

3.2.1 Simple pipeline scenario 1: a holding page

For your first app, and your first middleware pipeline, you'll learn how to create an app consisting of a holding page. This can be useful when you're first setting up your application, to ensure it's processing requests without errors.

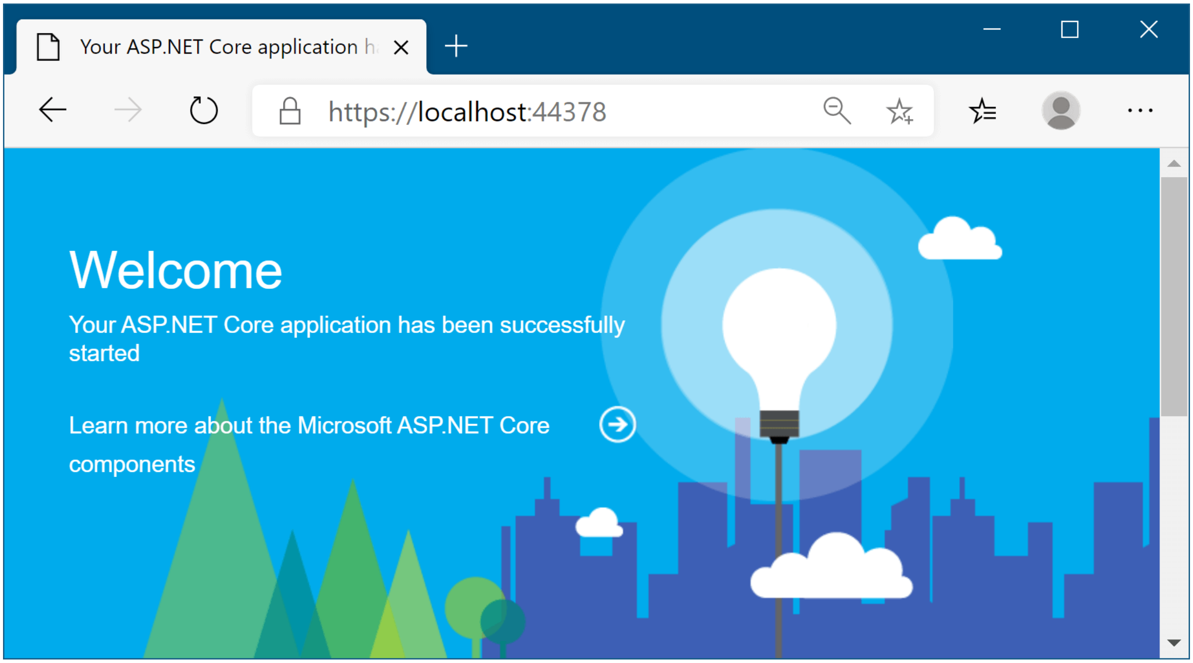
TIP Remember, you can view the application code for this book in the GitHub repository at https://github.com/andrewlock/asp-dot-net-core-in-action-2e.

In previous chapters, I've mentioned that the ASP.NET Core framework is composed of many small, individual libraries. You typically add a piece of middleware by referencing a package in your application's csproj project file and configuring the middleware in the `Configure` method of your `Startup` class. Microsoft ships many standard middleware components with ASP.NET Core for you to choose from, and you can also use third-party components from NuGet and GitHub, or you can build your own custom middleware.

NOTE I'll discuss building custom middleware in chapter 19.

Unfortunately, there isn't a definitive list of middleware available, but you can view the source code for all the middleware that comes as part of ASP.NET Core in the main ASP.NET Core GitHub repository (https://github.com/aspnet/aspnetcore). You can find most of the middleware in the src/Middleware folder, though some middleware is in other folders where it forms part of a larger feature. For example, the authentication and authorization middleware can be found in the src/Security folder instead. Alternatively, with a bit of searching on https://nuget.org you can often find middleware with the functionality you need.

In this section, you'll see how to create one of the simplest middleware pipelines, consisting of `WelcomePageMiddleware` only. `WelcomePageMiddleware` is designed to quickly provide a sample page when you're first developing an application, as you can see in figure 3.3. You wouldn't use it in a production app, as you can't customize the output, but it's a single, self-contained middleware component you can use to ensure your application is running correctly.



Welcome

Your ASP.NET Core application has been successfully started

Learn more about the Microsoft ASP.NET Core components

Figure 3.3 The Welcome page middleware response. Every request to the application, at any path, will return the same Welcome page response.

> **TIP** `WelcomePageMiddleware` **is included as part of the base ASP.NET Core framework, so you don't need to add a reference to any additional NuGet packages.**

Even though this application is simple, the exact same process occurs when it receives an HTTP request, as shown in figure 3.4.

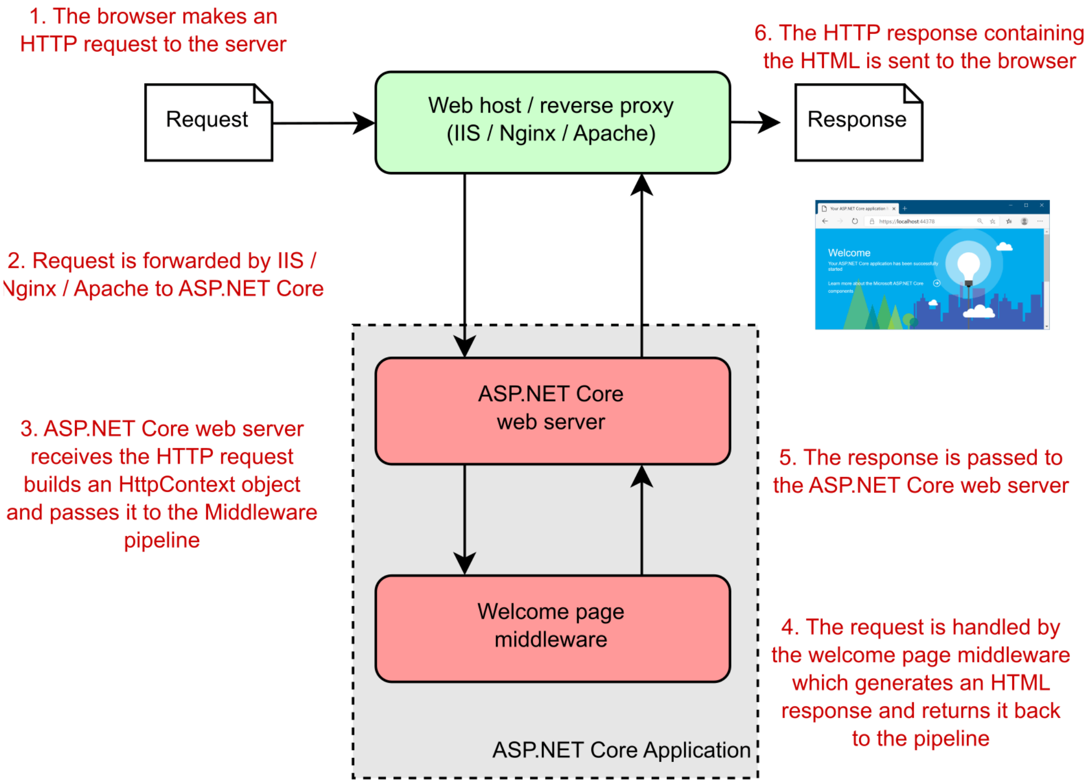

1. The browser makes an HTTP request to the server
Request
Web host / reverse proxy (IIS / Nginx / Apache)
6. The HTTP response containing the HTML is sent to the browser
Response
2. Request is forwarded by IIS / Nginx / Apache to ASP.NET Core
Welcome
ASP.NET Core web server
3. ASP.NET Core web server receives the HTTP request builds an HttpContext object and passes it to the Middleware pipeline
5. The response is passed to the ASP.NET Core web server
Welcome page middleware
4. The request is handled by the welcome page middleware which generates an HTML response and returns it back to the pipeline
ASP.NET Core Application

Figure 3.4 `WelcomePageMiddleware` **handles a request. The request passes from the reverse proxy to the ASP.NET Core web server and, finally, to the middleware pipeline, which generates an HTML response.**

The request passes to the ASP.NET Core web server, which builds a representation of the request and passes it to the middleware pipeline. As it’s the first (only!) middleware in the pipeline, `WelcomePageMiddleware` receives the request and must decide how to handle it. The middleware responds by generating an HTML response, no matter what request it receives. This response passes back to the ASP.NET Core web server, which forwards it on to the user to display in their browser.

As with all ASP.NET Core applications, you define the middleware pipeline in the `Configure` method of `Startup` by adding middleware to an `IApplicationBuilder` object. To create your first middleware pipeline, consisting of a single middleware component, you need just a single method call.

Listing 3.1 Startup for a Welcome page middleware pipeline

```
using Microsoft.AspNetCore.Builder;
namespace CreatingAHoldingPage
```

```
{
    public class Startup                                  #A
    {
        public void Configure(IApplicationBuilder app)    #B
        {
            app.UseWelcomePage();                         #C
        }
    }
}
```

#A The Startup class is very simple for this basic application.
#B The Configure method is used to define the middleware pipeline.
#C The only middleware in the pipeline

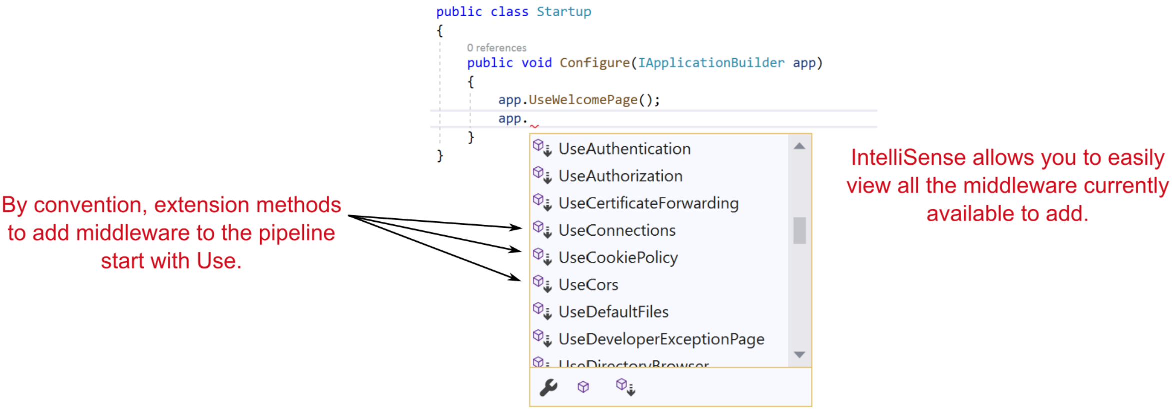
As you can see, the `Startup` for this application is very simple. The application has no configuration and no services, so `Startup` doesn't have a constructor or a `ConfigureServices` method. The only required method is `Configure`, in which you call `UseWelcomePage`.

You build the middleware pipeline in ASP.NET Core by calling methods on `IApplicationBuilder`, but this interface doesn't define methods like `UseWelcomePage` itself. Instead, these are *extension* methods.

Using extension methods allows you to effectively add functionality to the `IApplicationBuilder` class, while keeping their implementations isolated from it. Under the hood, the methods are typically calling *another* extension method to add the middleware to the pipeline. For example, behind the scenes, the `UseWelcomePage` method adds the `WelcomePageMiddleware` to the pipeline using

```
UseMiddleware<WelcomePageMiddleware>();
```

This convention of creating an extension method for each piece of middleware and starting the method name with `Use` is designed to improve discoverability when adding middleware to your application. ASP.NET Core includes a lot of middleware as part of the core framework, so you can use IntelliSense in Visual Studio and other IDEs to view all the middleware available, as shown in figure 3.5.



```
public class Startup
{
    0 references
    public void Configure(IApplicationBuilder app)
    {
        app.UseWelcomePage();
        app.
    }
}
```

By convention, extension methods to add middleware to the pipeline start with Use.

IntelliSense allows you to easily view all the middleware currently available to add.

Figure 3.5 IntelliSense makes it easy to view all the middleware available to add to your middleware pipeline.

Calling the `UseWelcomePage` method adds the `WelcomePageMiddleware` as the next middleware in the pipeline. Although you're only using a single middleware component here, it's important to remember that the order in which you make calls to `IApplicationBuilder` in `Configure` defines the order that the middleware will run in the pipeline.

> **WARNING** Always take care when adding middleware to the pipeline and consider the order in which it will run. A component can only access data created by middleware that comes before it in the pipeline.

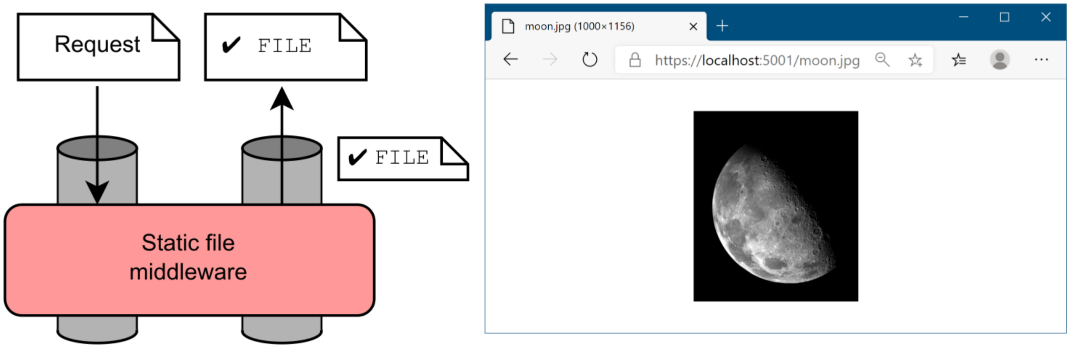
This is the most basic of applications, returning the same response no matter which URL you navigate to, but it shows how easy it is to define your application behavior using middleware. Now we'll make things a little more interesting and return a different response when you make requests to different paths.

3.2.2 Simple pipeline scenario 2: Handling static files

In this section, I'll show how to create one of the simplest middleware pipelines you can use for a full application: a static file application.

Most web applications, including those with dynamic content, serve a number of pages using static files. Images, JavaScript, and CSS stylesheets are normally saved to disk during development and are served up when requested, normally as part of a full HTML page request.

For now, you'll use `StaticFileMiddleware` to create an application that only serves static files from the wwwroot folder when requested, as shown in figure 3.6. In this example, an image called moon.jpg exists in the wwwroot folder. When you request the file using the `/moon.jpg` path, it's loaded and returned as the response to the request.



1. The static file middleware handles the request by returning the file requested

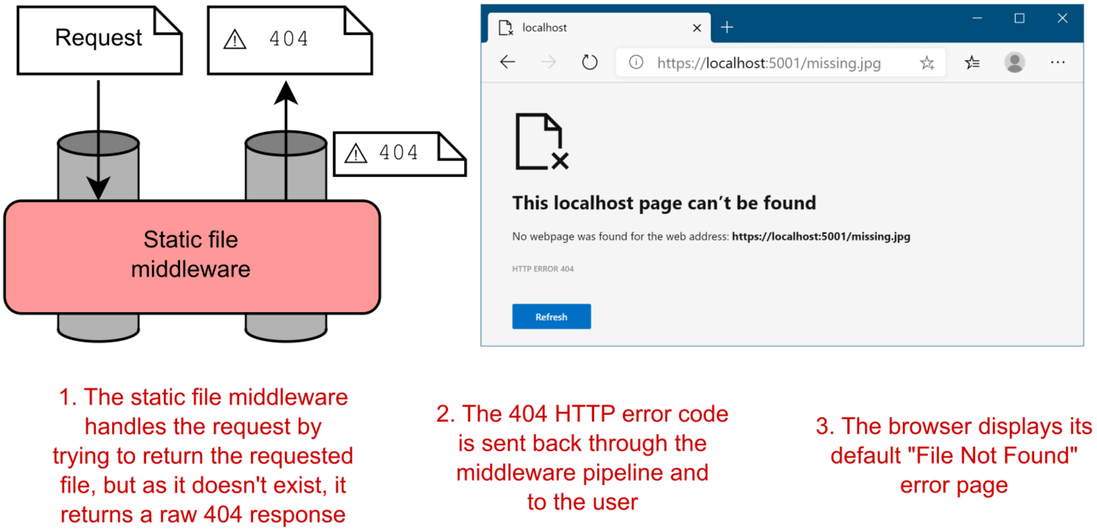
2. The file stream is sent back through the middleware pipeline, and out to the browser

3. The browser displays the file returned in the response

Figure 3.6 Serving a static image file using the static file middleware

If the user requests a file that doesn't exist in the wwwroot folder, for example missing.jpg, then the static file middleware won't serve a file. Instead, a 404 HTTP error code response will be sent to the user's browser, which will show its default "File Not Found" page, as shown in figure 3.7.

> **NOTE** How this page looks will depend on your browser. In some browsers, for example Internet Explorer (IE), you might see a completely blank page.



This localhost page can’t be found

No webpage was found for the web address: **https://localhost:5001/missing.jpg**

HTTP ERROR 404

1. The static file middleware handles the request by trying to return the requested file, but as it doesn't exist, it returns a raw 404 response

2. The 404 HTTP error code is sent back through the middleware pipeline and to the user

3. The browser displays its default "File Not Found" error page

Figure 3.7 Returning a 404 to the browser when a file doesn’t exist. The requested file did not exist in the wwwroot folder, so the ASP.NET Core application returned a 404 response. The browser, Microsoft Edge in this case, will then show the user a default “File Not Found” error.

Building the middleware pipeline for this application is easy, consisting of a single piece of middleware, `StaticFileMiddleware`, as you can see in the following listing. You don’t need any services, so configuring the middleware pipeline in `Configure` with `UseStaticFiles` is all that’s required.

Listing 3.2 Startup for a static file middleware pipeline

```
using Microsoft.AspNetCore.Builder;

namespace CreatingAStaticFileWebsite
{
    public class Startup                                  #A
    {
        public void Configure(IApplicationBuilder app)    #B
        {
            app.UseStaticFiles();                         #C
        }
    }
```

#A The Startup class is very simple for this basic static file application.
#B The Configure method is used to define the middleware pipeline.
#C The only middleware in the pipeline

TIP Remember, you can view the application code for this book in the GitHub repository at https://github.com/andrewlock/asp-dot-net-core-in-action-2e.

When the application receives a request, the ASP.NET Core web server handles it and passes it to the middleware pipeline. `StaticFileMiddleware` receives the request and determines whether or not it can handle it. If the requested file exists, the middleware handles the request and returns the file as the response, as shown in figure 3.8.

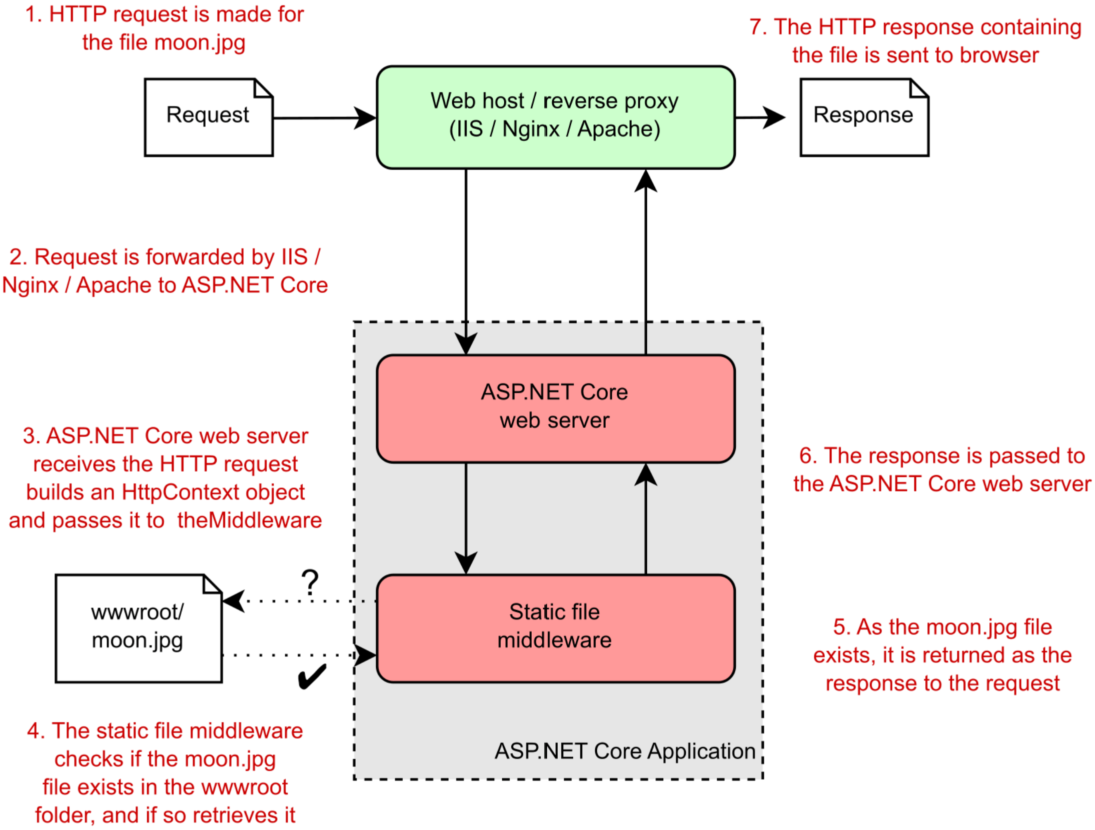

1. HTTP request is made for the file moon.jpg
Request
Web host / reverse proxy (IIS / Nginx / Apache)
7. The HTTP response containing the file is sent to browser
Response
2. Request is forwarded by IIS / Nginx / Apache to ASP.NET Core
ASP.NET Core web server
3. ASP.NET Core web server receives the HTTP request builds an HttpContext object and passes it to theMiddleware
6. The response is passed to the ASP.NET Core web server
wwwroot/ moon.jpg
?
✔
Static file middleware
5. As the moon.jpg file exists, it is returned as the response to the request
4. The static file middleware checks if the moon.jpg file exists in the wwwroot folder, and if so retrieves it
ASP.NET Core Application

Figure 3.8 `StaticFileMiddleware` **handles a request for a file. The middleware checks the wwwroot folder to see if the requested moon.jpg file exists. The file exists, so the middleware retrieves it and returns it as the response to the web server and, ultimately, out to the browser.**

If the file doesn't exist, then the request effectively passes *through* the static file middleware unchanged. But wait, you only added one piece of middleware, right? Surely you can't pass the request through to the next middleware if there *isn't* another one?

ASP.NET Core automatically adds an "dummy" piece of middleware to the end of the pipeline. This middleware always returns a 404 response if it's called.

TIP Remember, if no middleware generates a response for a request, the pipeline will automatically return a simple 404 error response to the browser.

HTTP response status codes

Every HTTP response contains a *status code* and, optionally, a *reason phrase* describing the status code. Status codes are fundamental to the HTTP protocol and are a standardized way of indicating common results. A 200 response, for example, means the request was successfully answered, whereas a 404 response indicates that the resource requested couldn't be found.

Status codes are always three digits long and are grouped into five different classes, based on the first digit:

- *1xx*—Information. Not often used, provides a general acknowledgment.
- *2xx*—Success. The request was successfully handled and processed.
- *3xx*—Redirection. The browser must follow the provided link, to allow the user to log in, for example.
- *4xx*—Client error. There was a problem with the request. For example, the request sent invalid data, or the user isn't authorized to perform the request.
- *5xx*—Server error. There was a problem on the server that caused the request to fail.

These status codes typically drive the behavior of a user's browser. For example, the browser will handle a 301 response automatically, by redirecting to the provided new link and making a second request, all without the user's interaction.

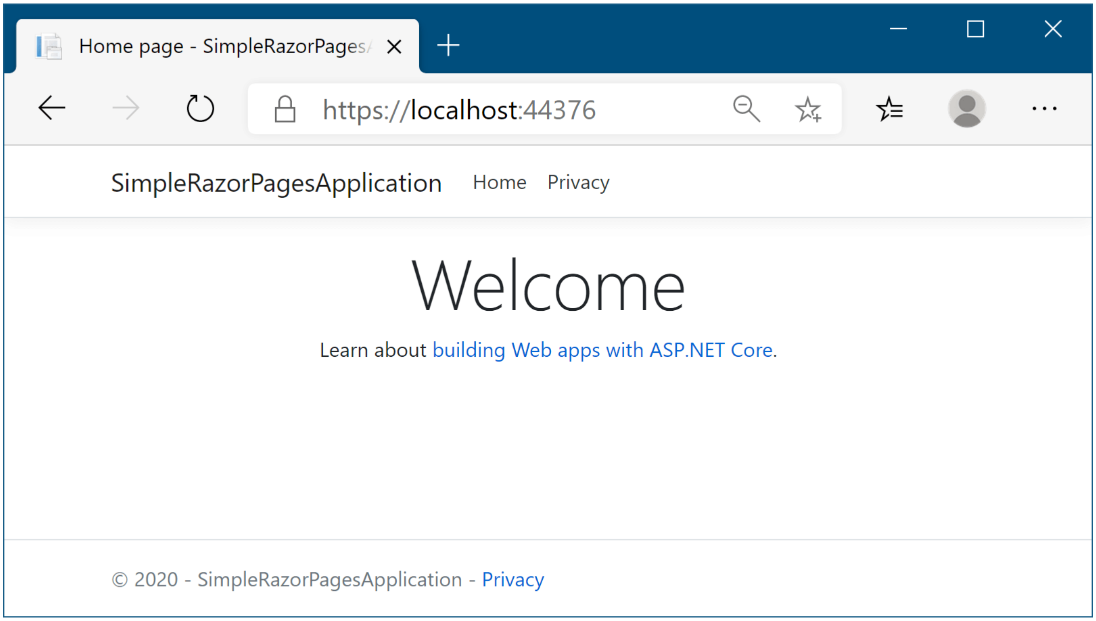
Error codes are found in the 4xx and 5xx classes. Common codes include a 404 response when a file couldn't be found, a 400 error when a client sends invalid data (an invalid email address for example), and a 500 error when an error occurs on the server. HTTP responses for error codes may or may not include a response body, which is content to display when the client receives the response.

This basic ASP.NET Core application allows you to easily see the behavior of the ASP.NET Core middleware pipeline and the static file middleware in particular, but it's unlikely your applications will be as simple as this. It's more likely that static files will form one part of your middleware pipeline. In the next section, we'll look at how to combine multiple middleware, looking at a simple Razor Pages application.

3.2.3 Simple pipeline scenario 3: A Razor Pages application

By this point, you, I hope, have a decent grasp of the middleware pipeline, insofar as understanding that it defines your application's behavior. In this section, you'll see how to combine multiple middleware to form a pipeline, using several standard middleware components. As before, this is performed in the `Configure` method of `Startup` by adding middleware to an `IApplicationBuilder` object.

You'll begin by creating a basic middleware pipeline that you'd find in a typical ASP.NET Core Razor Pages template and then extend it by adding middleware. The output when you navigate to the homepage of the application is shown in figure 3.9—identical to the sample application shown in chapter 2.



Welcome

Learn about building Web apps with ASP.NET Core.

© 2020 - SimpleRazorPagesApplication - Privacy

Figure 3.9 A simple Razor Pages application. The application uses only four pieces of middleware: routing middleware to choose a Razor Page to run, endpoint middleware to generate the HTML from a Razor Page, static file middleware to serve the CSS files, and an exception handler middleware to capture any errors.

Creating this application requires only four pieces of middleware: routing middleware to choose a Razor Page to execute, endpoint middleware to generate the HTML from a Razor Page, static file middleware to serve the CSS and image files from the wwwroot folder, and an exception handler middleware to handle any errors that might occur.

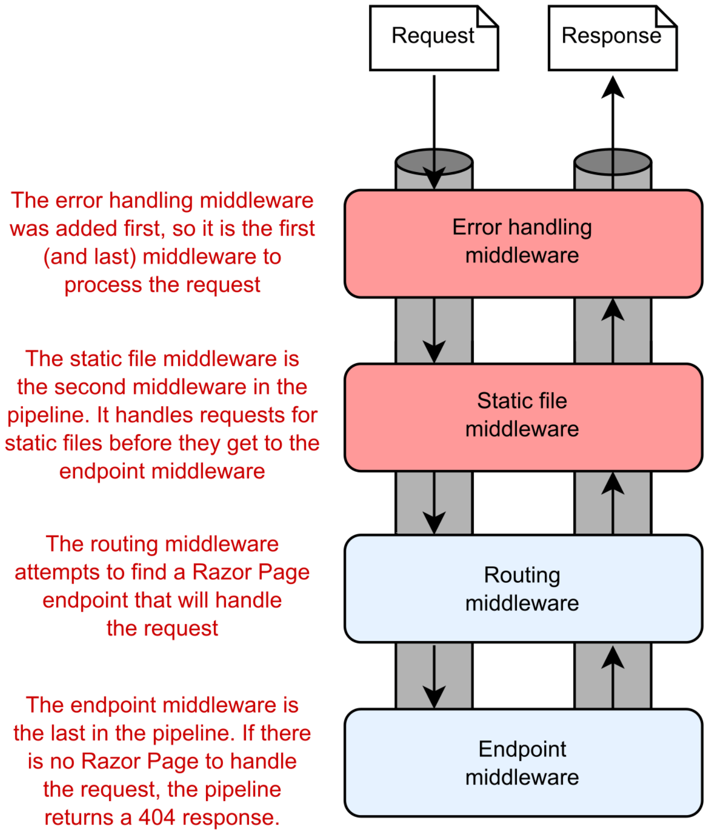
The configuration of the middleware pipeline for the application occurs in the `Configure` method of `Startup`, as always, and is shown in the following listing. As well as the middleware configuration, this also shows the call to `AddRazorPages()` in `ConfigureServices`, which is required when using Razor Pages. You'll learn more about service configuration in chapter 10.

Listing 3.3 A basic middleware pipeline for a Razor Pages application

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services
    {
        services.AddRazorPages();
    }
    public void Configure(IApplicationBuilder app)
    {
        app.UseExceptionHandler("/Error");
        app.UseStaticFiles();
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

The addition of middleware to IApplicationBuilder to form the pipeline should be familiar to you now, but there are a couple of points worth noting in this example. First, all of the methods for adding middleware all start with Use. As I mentioned earlier, this is thanks to the convention of using extension methods to extend the functionality of IApplicationBuilder; by prefixing the methods with Use they should be easier to discover.

Another important point about this listing is the order of the Use methods in the Configure method. The order in which you add the middleware to the IApplicationBuilder object is the same order in which they're added to the pipeline. This creates a pipeline similar to that shown in figure 3.10.



The error handling middleware was added first, so it is the first (and last) middleware to process the request

The static file middleware is the second middleware in the pipeline. It handles requests for static files before they get to the endpoint middleware

The routing middleware attempts to find a Razor Page endpoint that will handle the request

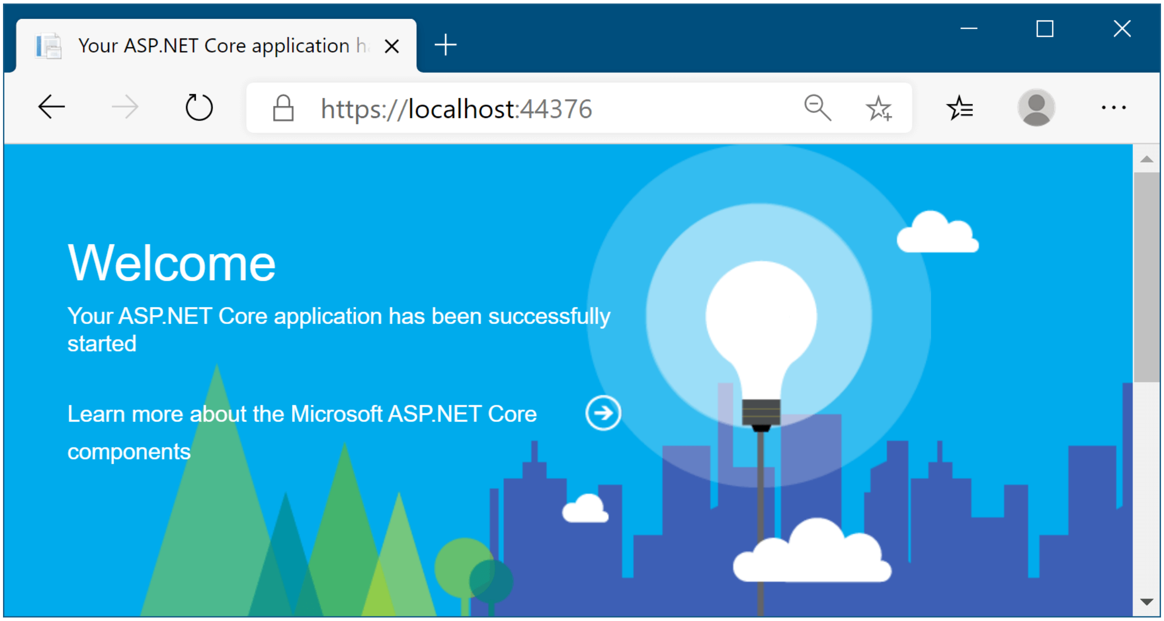
The endpoint middleware is the last in the pipeline. If there is no Razor Page to handle the request, the pipeline returns a 404 response.

Figure 3.10 The middleware pipeline for the example application in listing 3.3. The order in which you add the middleware to `IApplicationBuilder` defines the order of the middleware in the pipeline.

The exception handler middleware is called first, which passes the request on to the static file middleware. The static file handler will generate a response if the request corresponds to a file, otherwise it will pass the request on to the routing middleware. The routing middleware selects a Razor page based on the request URL, and the endpoint middleware executes the selected Razor Page. If no Razor Page can handle the requested URL, the automatic dummy middleware returns a 404 response.

> **NOTE** In versions 1.x and 2.x of ASP.NET Core, the routing and endpoint middleware were combined into a single “MVC” middleware. Splitting the responsibilities for routing from execution makes it possible to insert middleware *between* the routing and endpoint middleware. I discuss routing further in chapter 5.

The impact of ordering can most obviously be seen when you have two pieces of middleware that are both listening for the same path. For example, the endpoint middleware in the example pipeline currently responds to a request to the homepage of the application (with the "/" path) by generating the HTML response shown previously in figure 3.9. Figure 3.11 shows what happens if you reintroduce a piece of middleware you saw previously, WelcomePageMiddleware, and configure it to respond to the "/" path as well.



Welcome

Your ASP.NET Core application has been successfully started

Learn more about the Microsoft ASP.NET Core components

Figure 3.11 The Welcome page middleware response. The Welcome page middleware comes before the endpoint middleware, so a request to the homepage returns the Welcome page middleware instead of the Razor Pages response.

As you saw in section 3.2.1, WelcomePageMiddleware is designed to return a fixed HTML response, so you wouldn't use it in a production app, but it illustrates the point nicely. In the following listing, it's added to the start of the middleware pipeline and configured to respond only to the "/" path.

Listing 3.4 Adding `WelcomePageMiddleware` to the pipeline

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services
    {
        services.AddRazorPages();
    }
    public void Configure(IApplicationBuilder app)
    {
        app.UseWelcomePage("/");                       #A
        app.UseExceptionHandler("/Error");
        app.UseStaticFiles();
        app.UseRouting();
        app.UseEndpoints(endpoints =>                  #B
        {
            endpoints.MapRazorPages();
        });
    }
}
```

#A WelcomePageMiddleware handles all requests to the "/" path and returns a sample HTML response.
#B Requests to "/" will never reach the endpoint middleware.

Even though you know the endpoint middleware can also handle the "/" path, `WelcomePageMiddleware` is earlier in the pipeline, so it returns a response when it receives the request to "/", short-circuiting the pipeline, as shown in figure 3.12. None of the other middleware in the pipeline runs for the request, so none has an opportunity to generate a response.

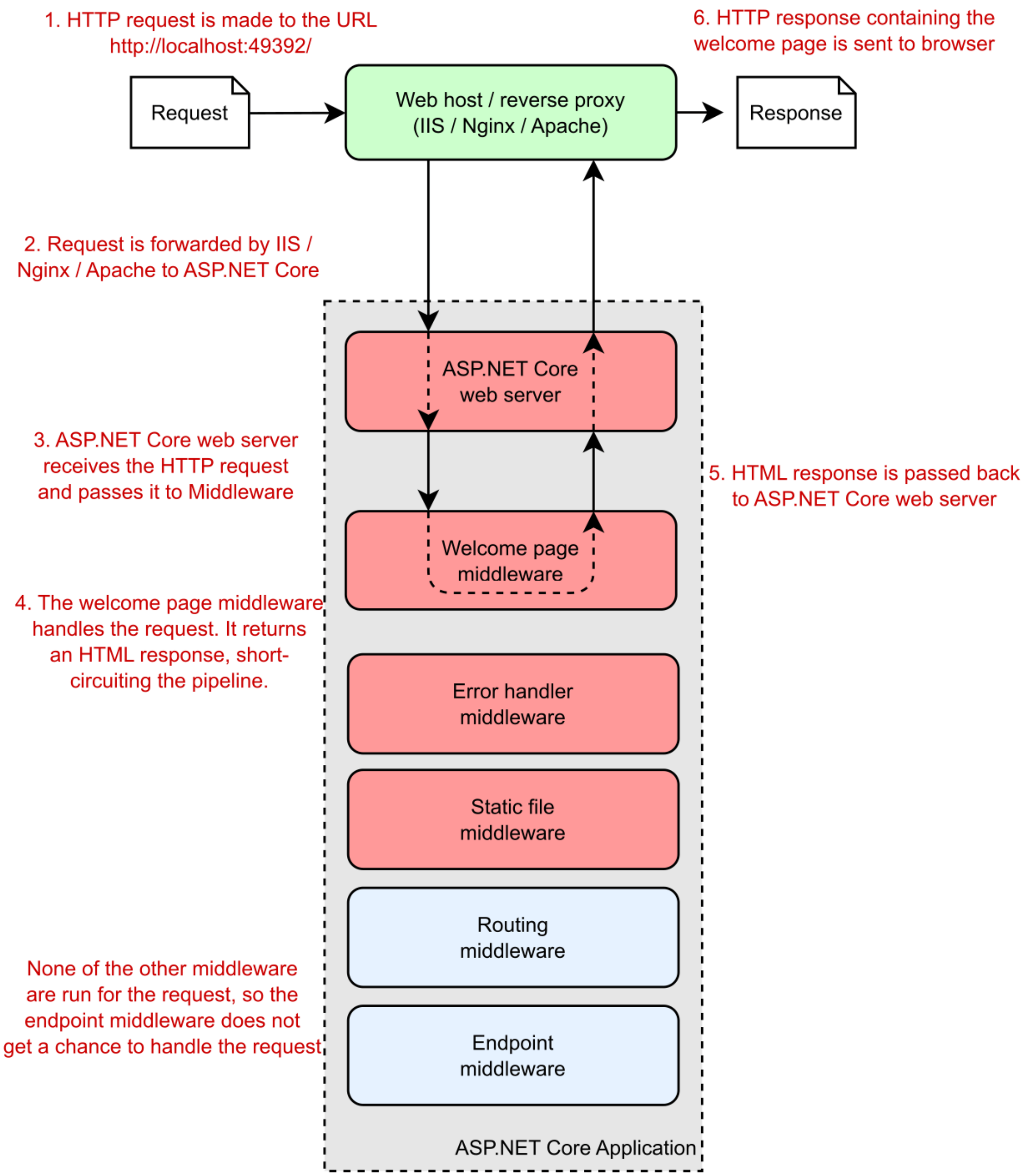

1. HTTP request is made to the URL http://localhost:49392/
6. HTTP response containing the welcome page is sent to browser
Request
Web host / reverse proxy (IIS / Nginx / Apache)
Response
2. Request is forwarded by IIS / Nginx / Apache to ASP.NET Core
ASP.NET Core web server
3. ASP.NET Core web server receives the HTTP request and passes it to Middleware
5. HTML response is passed back to ASP.NET Core web server
Welcome page middleware
4. The welcome page middleware handles the request. It returns an HTML response, short-circuiting the pipeline.
Error handler middleware
Static file middleware
Routing middleware
None of the other middleware are run for the request, so the endpoint middleware does not get a chance to handle the request
Endpoint middleware
ASP.NET Core Application

Figure 3.12 Overview of the application handling a request to the "/" path. The welcome page middleware is first in the middleware pipeline, so it receives the request before any other middleware. It generates an HTML response, short-circuiting the pipeline. No other middleware runs for the request.

If you moved WelcomePageMiddleware to the end of the pipeline, after the call to UseEndpoints, then you'd have the opposite situation. Any requests to "/" would be handled by the endpoint middleware and you'd never see the Welcome page.

> **TIP** You should always consider the order of middleware when adding to the Configure method. Middleware added earlier in the pipeline will run (and potentially return a response) before middleware added later.

All the examples shown so far attempt to handle an incoming request and generate a response, but it's important to remember that the middleware pipeline is bi-directional. Each middleware component gets an opportunity to handle both the incoming request and the outgoing response. The order of middleware is most important for those components that create or modify the outgoing response.

In the previous example, I included ExceptionHandlerMiddleware at the start of the application's middleware pipeline, but it didn't seem to do anything. Error handling middleware characteristically ignores the incoming request as it arrives in the pipeline, and instead inspects the outgoing response, only modifying it when an error has occurred. In the next section, I'll detail the types of error handling middleware that are available to use with your application and when to use them.

3.3 Handling errors using middleware

Errors are a fact of life when developing applications. Even if you write perfect code, as soon as you release and deploy your application, users will find a way to break it, whether by accident or intentionally! The important thing is that your application handles these errors gracefully, providing a suitable response to the user, and doesn't cause your whole application to fail.

The design philosophy for ASP.NET Core is that every feature is opt-in. So, as error handling is a feature, you need to explicitly enable it in your application. Many different types of errors could occur in your application and there are many different ways to handle them, but in this section I'll focus on two: exceptions and error status codes.

Exceptions typically occur whenever you find an unexpected circumstance. A typical (and highly frustrating) exception you'll no doubt have experienced before is NullReferenceException, which is thrown when you attempt to access an object that hasn't been initialized[14]. If an exception occurs in a middleware component, it propagates up the pipeline, as shown in figure 3.13. If the pipeline doesn't handle the exception, the web server will return a 500 status code back to the user.

[14] C# 8.0 introduced non-nullable reference types. These provide a way to more clearly handle null values, with the promise of finally ridding .NET of NullReferenceExceptions! It will take a while for that to come true, as the feature is opt-in, and requires library support, but the future's bright. See the documentation to get started: https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/nullable-reference-types

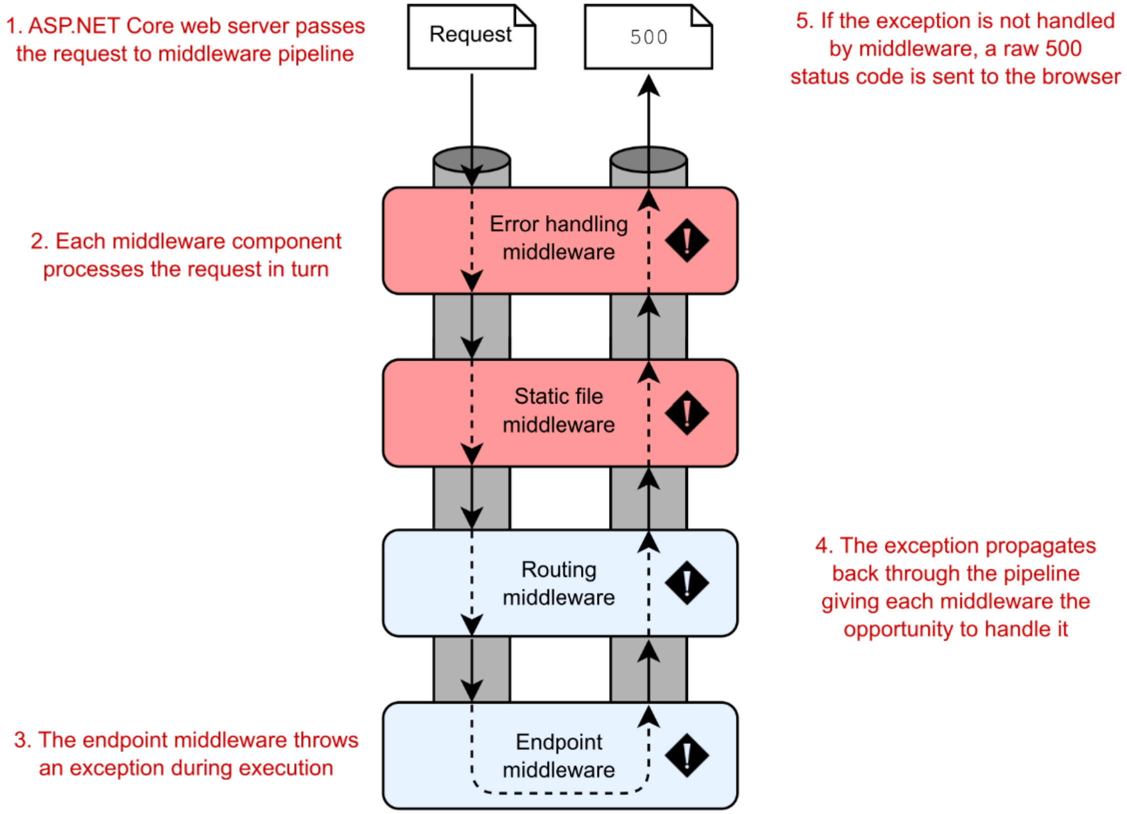
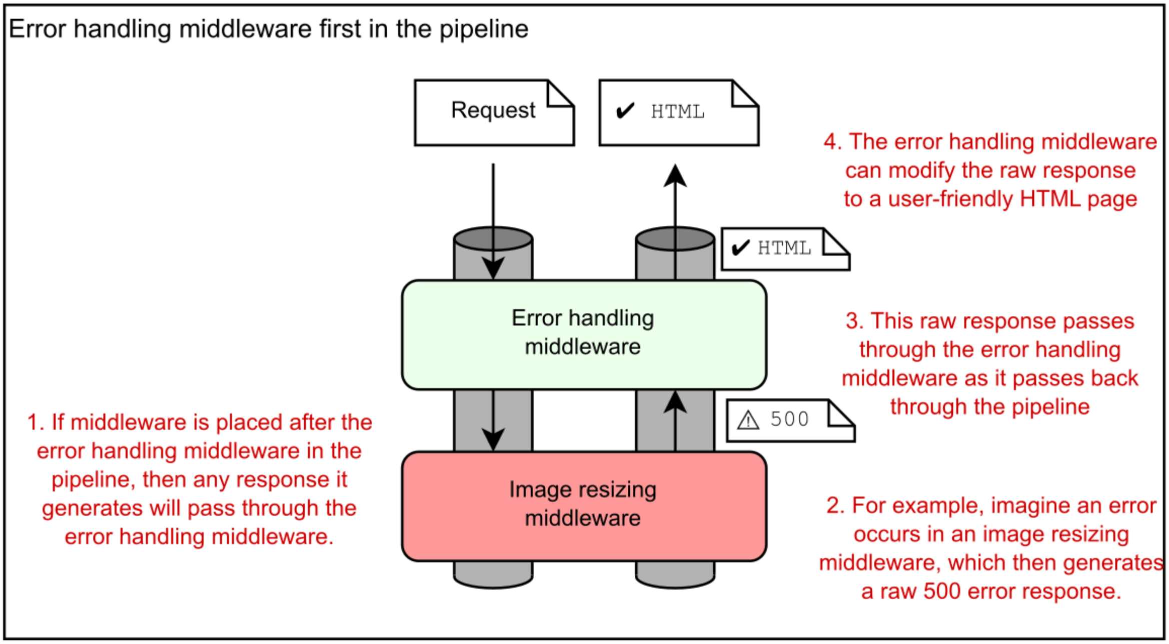

1. ASP.NET Core web server passes the request to middleware pipeline
Request
500
5. If the exception is not handled by middleware, a raw 500 status code is sent to the browser
2. Each middleware component processes the request in turn
Error handling middleware
Static file middleware
Routing middleware
4. The exception propagates back through the pipeline giving each middleware the opportunity to handle it
Endpoint middleware
3. The endpoint middleware throws an exception during execution

Figure 3.13 An exception in the endpoint middleware propagates through the pipeline. If the exception isn't caught by middleware earlier in the pipeline, then a 500 "Server error" status code is sent to the user's browser.

In some situations, an error won't cause an exception. Instead, middleware might generate an error status code. One such case you've already seen is when a requested path isn't handled. In that situation, the pipeline will return a 404 error, which results in a generic, unfriendly page being shown to the user, as you saw in figure 3.7. Although this behavior is "correct," it doesn't provide a great experience for users of your application.

Error handling middleware attempts to address these problems by modifying the response before the app returns it to the user. Typically, error handling middleware either returns details of the error that occurred, or it returns a generic, but friendly, HTML page to the user. You should always place error handling middleware early in the middleware pipeline to ensure it will catch any errors generated in subsequent middleware, as shown in figure 3.14. Any responses generated by middleware earlier in the pipeline than the error handling middleware can't be intercepted.



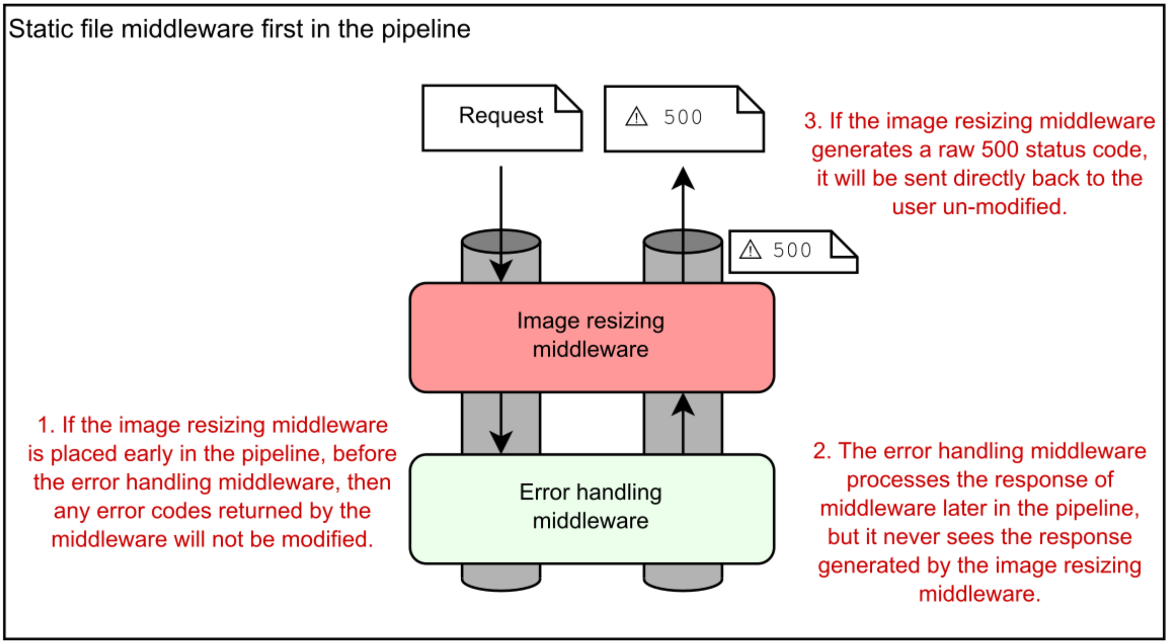
Error handling middleware first in the pipeline

4. The error handling middleware can modify the raw response to a user-friendly HTML page

3. This raw response passes through the error handling middleware as it passes back through the pipeline

1. If middleware is placed after the error handling middleware in the pipeline, then any response it generates will pass through the error handling middleware.

2. For example, imagine an error occurs in an image resizing middleware, which then generates a raw 500 error response.



Static file middleware first in the pipeline

3. If the image resizing middleware generates a raw 500 status code, it will be sent directly back to the user un-modified.

1. If the image resizing middleware is placed early in the pipeline, before the error handling middleware, then any error codes returned by the middleware will not be modified.

2. The error handling middleware processes the response of middleware later in the pipeline, but it never sees the response generated by the image resizing middleware.

Figure 3.14 Error handling middleware should be placed early in the pipeline to catch raw status code errors. In the first case, the error handling middleware is placed before the image resizing middleware, so it can replace raw status code errors with a user-friendly error page. In the second case, the error handling middleware is placed after the image resizing middleware, so raw error status codes can't be modified.

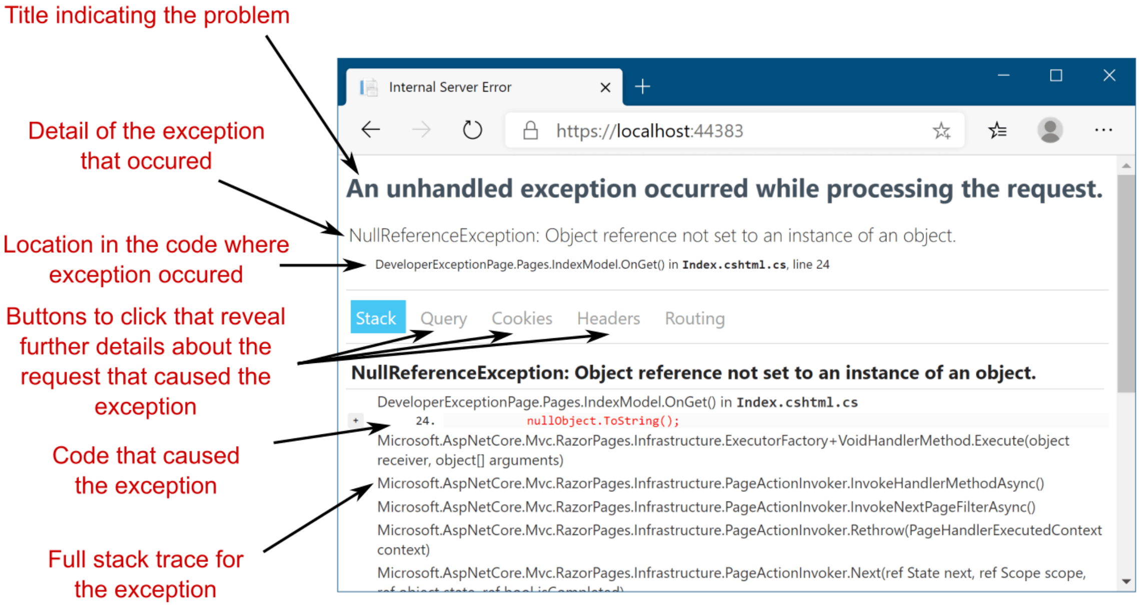
The remainder of this section shows several types of error handling middleware that are available for use in your application. They are available as part of the base ASP.NET Core framework, so you don't need to reference any additional NuGet packages to use them.

3.3.1 Viewing exceptions in development: DeveloperExceptionPage

When you're developing an application, you typically want access to as much information as possible when an error occurs somewhere in your app. For that reason, Microsoft provides `DeveloperExceptionPageMiddleware`, which can be added to your middleware pipeline using

```
app.UseDeveloperExceptionPage();
```

When an exception is thrown and propagates up the pipeline to this middleware, it will be captured. The middleware then generates a friendly HTML page, which it returns with a 500 status code to the user, as shown in figure 3.15. This page contains a variety of details about the request and the exception, including the exception stack trace, the source code at the line the exception occurred, and details of the request, such as any cookies or headers that had been sent.


Title indicating the problem
Detail of the exception that occured
Location in the code where exception occured
Buttons to click that reveal further details about the request that caused the exception
Code that caused the exception
Full stack trace for the exception
Internal Server Error
https://localhost:44383
Stack
Query
Cookies
Headers
Routing

An unhandled exception occurred while processing the request.

NullReferenceException: Object reference not set to an instance of an object.

DeveloperExceptionPage.Pages.IndexModel.OnGet() in `Index.cshtml.cs`, line 24

NullReferenceException: Object reference not set to an instance of an object.

DeveloperExceptionPage.Pages.IndexModel.OnGet() in `Index.cshtml.cs`

```
24.         nullObject.ToString();
```

Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.ExecutorFactory+VoidHandlerMethod.Execute(object receiver, object[] arguments)

Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker.InvokeHandlerMethodAsync()

Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker.InvokeNextPageFilterAsync()

Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker.Rethrow(PageHandlerExecutedContext context)

Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker.Next(ref State next, ref Scope scope,

Figure 3.15 The developer exception page shows details about the exception when it occurs during the process of a request. The location in the code that caused the exception, the source code line itself, and the stack trace are all shown by default. You can also click the Query, Cookies, Headers, or Routing buttons to reveal further details about the request that caused the exception.

Having these details available when an error occurs is invaluable for debugging a problem, but they also represent a security risk if used incorrectly. You should never return more details about your application to users than absolutely necessary, so you should only ever use `DeveloperExceptionPage` when developing your application. The clue is in the name!

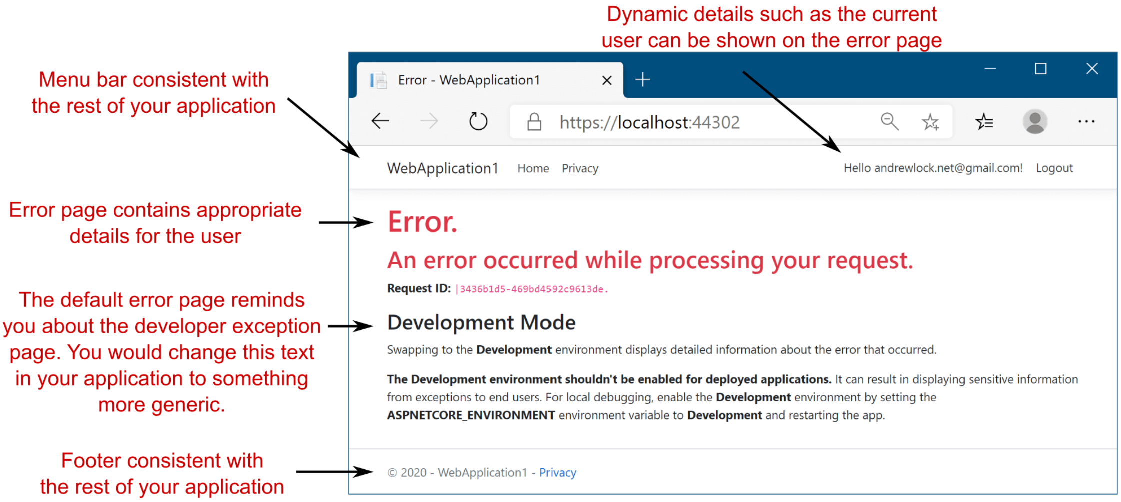
> **WARNING** **Never use the developer exception page when running in production. Doing so is a security risk as it could publicly reveal details about your application's code, making you an easy target for attackers.**

If the developer exception page isn't appropriate for production use, what should you use instead? Luckily, there's another general-purpose error handling middleware you *can* use in production, one that you've already seen and used: `ExceptionHandlerMiddleware`.

3.3.2 Handling exceptions in production: ExceptionHandlerMiddleware

The developer exception page is handy when developing your applications, but you shouldn't use it in production as it can leak information about your app to potential attackers. You still want to catch errors though, otherwise users will see unfriendly error pages or blank pages, depending on the browser they're using.

You can solve this problem by using `ExceptionHandlerMiddleware`. If an error occurs in your application, the user will be presented with a custom error page that's consistent with the rest of the application, but that only provides the necessary details about the error. For example, a custom error page, such as the one shown in figure 3.16, can keep the look and feel of the application by using the same header, displaying the currently logged-in user, and only displaying an appropriate message to the user instead of the full details of the exception.



Dynamic details such as the current user can be shown on the error page

Menu bar consistent with the rest of your application

Error page contains appropriate details for the user

The default error page reminds you about the developer exception page. You would change this text in your application to something more generic.

Footer consistent with the rest of your application

Error.

An error occurred while processing your request.

Request ID: |3436b1d5-469bd4592c9613de.

Development Mode

Swapping to the **Development** environment displays detailed information about the error that occurred.

The Development environment shouldn't be enabled for deployed applications. It can result in displaying sensitive information from exceptions to end users. For local debugging, enable the **Development** environment by setting the **ASPNETCORE_ENVIRONMENT** environment variable to **Development** and restarting the app.

© 2020 - WebApplication1 - Privacy

Figure 3.16 A custom error page created by `ExceptionHandlerMiddleware`. The custom error page can keep the same look and feel as the rest of the application by reusing elements such as the header and footer. More importantly, you can easily control the error details displayed to users.

If you were to peek at the `Configure` method of almost any ASP.NET Core application, you'd almost certainly find the developer exception page used in combination with `ExceptionHandlerMiddleware`, in a similar manner to that shown here.

Listing 3.5 Configuring exception handling for development and production

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())                            #A
    {
        app.UseDeveloperExceptionPage();                #B
    }
    else
    {
        app.UseExceptionHandler("/Error");             #C
    }

    // additional middleware configuration
}
```

#A Configure a different pipeline when running in development.
#B The developer exception page should only be used when running in development mode.
#C When in production, ExceptionHandlerMiddleware is added to the pipeline.

As well as demonstrating how to add `ExceptionHandlerMiddleware` to your middleware pipeline, this listing shows that it's perfectly acceptable to configure different middleware pipelines depending on the environment when the application starts up. You could also vary your pipeline based on other values, such as settings loaded from configuration.

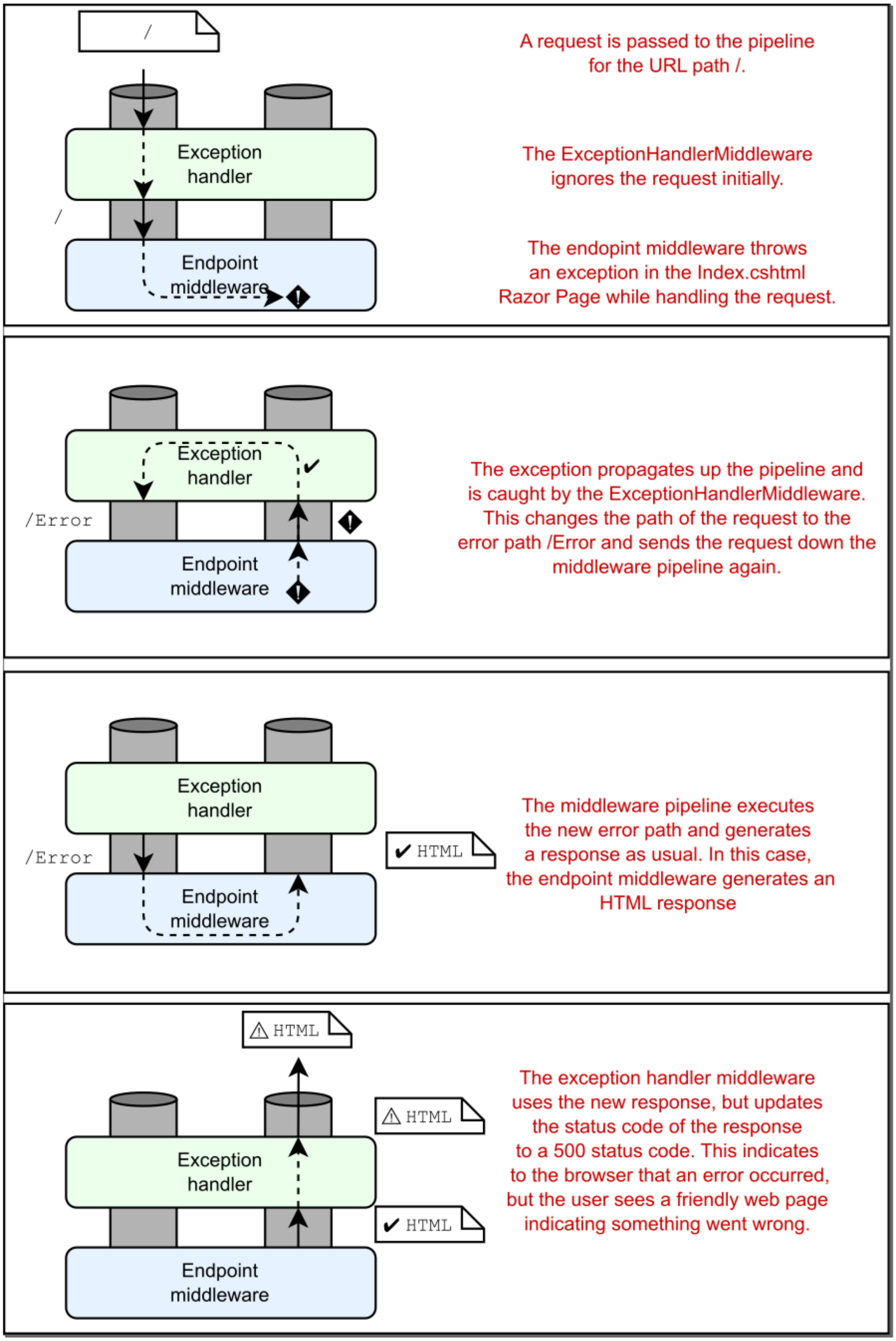
NOTE **You'll see how to use configuration values to customize the middleware pipeline in chapter 11.**

When adding `ExceptionHandlerMiddleware` to your application, you'll typically provide a path to the custom error page that will be displayed to the user. In the example listing, you used an error handling path of `"/Error"`:

```
app.UseExceptionHandler("/Error");
```

`ExceptionHandlerMiddleware` will invoke this path after it captures an exception, in order to generate the final response. The ability to dynamically generate a response is a key feature of `ExceptionHandlerMiddleware`—it allows you to re-execute a middleware pipeline in order to generate the response sent to the user.

Figure 3.17 shows what happens when `ExceptionHandlerMiddleware` handles an exception. It shows the flow of events when the Index.chstml Razor Page generates an exception when a request is made to the `"/"` path. The final response returns an error status code but also provides an HTML response to display to the user, using the `"/Error"` path.



A request is passed to the pipeline for the URL path /.

The ExceptionHandlerMiddleware ignores the request initially.

The endopint middleware throws an exception in the Index.cshtml Razor Page while handling the request.

The exception propagates up the pipeline and is caught by the ExceptionHandlerMiddleware. This changes the path of the request to the error path /Error and sends the request down the middleware pipeline again.

The middleware pipeline executes the new error path and generates a response as usual. In this case, the endpoint middleware generates an HTML response

The exception handler middleware uses the new response, but updates the status code of the response to a 500 status code. This indicates to the browser that an error occurred, but the user sees a friendly web page indicating something went wrong.

Figure 3.17 `ExceptionHandlerMiddleware` **handling an exception to generate an HTML response. A request to the / path generates an exception, which is handled by the middleware. The pipeline is re-executed using the /Error path to generate the HTML response.**

The sequence of events when an exception occurs somewhere in the middleware pipeline after `ExceptionHandlerMiddleware` is as follows:

1. A piece of middleware throws an exception.
2. `ExceptionHandlerMiddleware` catches the exception.
3. Any partial response that has been defined is cleared.
4. The middleware overwrites the request path with the provided error handling path.
5. The middleware sends the request back down the pipeline, as though the original request had been for the error handling path.
6. The middleware pipeline generates a new response as normal.
7. When the response gets back to `ExceptionHandlerMiddleware`, it modifies the status code to a 500 error and continues to pass the response up the pipeline to the web server.

The main advantage that re-executing the pipeline brings is the ability to have your error messages integrated into your normal site layout, as shown previously in figure 3.16. It's certainly possible to return a fixed response when an error occurs, but you wouldn't be able to have a menu bar with dynamically generated links or display the current user's name in the menu. By re-executing the pipeline, you can ensure that all the dynamic areas of your application are correctly integrated, as if the page was a standard page of your site.

> **NOTE** You don't need to do anything other than add `ExceptionHandlerMiddleware` to your application and configure a valid error handling path to enable re-executing the pipeline. The middleware will catch the exception and re-execute the pipeline for you. Subsequent middleware will treat the re-execution as a new request, but previous middleware in the pipeline won't be aware anything unusual happened.

Re-executing the middleware pipeline is a great way to keep consistency in your web application for error pages, but there are some gotchas to be aware of. First, middleware can only modify a response generated further down the pipeline if the response *hasn't yet been sent to the client*. This can be a problem if, for example, an error occurs while ASP.NET Core is sending a static file to a client. In that case, where bytes have already begun to be sent, the error handling middleware won't be able to run, as it can't reset the response. Generally speaking, there's not a lot you can do about this issue, but it's something to be aware of.

A more common problem occurs when the error handling path throws an error during the re-execution of the pipeline. Imagine there's a bug in the code that generates the menu at the top of the page:

1. When the user reaches your homepage, the code for generating the menu bar throws an exception.
2. The exception propagates up the middleware pipeline.

3. When reached, `ExceptionHandlerMiddleware` captures it and the pipe is re-executed using the error handling path.
4. When the error page executes, it attempts to generate the menu bar for your app, which again throws an exception.
5. The exception propagates up the middleware pipeline.
6. `ExceptionHandlerMiddleware` has already tried to intercept a request, so it will let the error propagate all the way to the top of the middleware pipeline.
7. The web server returns a raw 500 error, as though there was no error handling middleware at all.

Thanks to this problem, it's often good practice to make your error handling pages as simple as possible, to reduce the possibility of errors occurring.

> **WARNING** If your error handling path generates an error, the user will see a generic browser error. It's often better to use a static error page that will always work, rather than a dynamic page that risks throwing more errors.

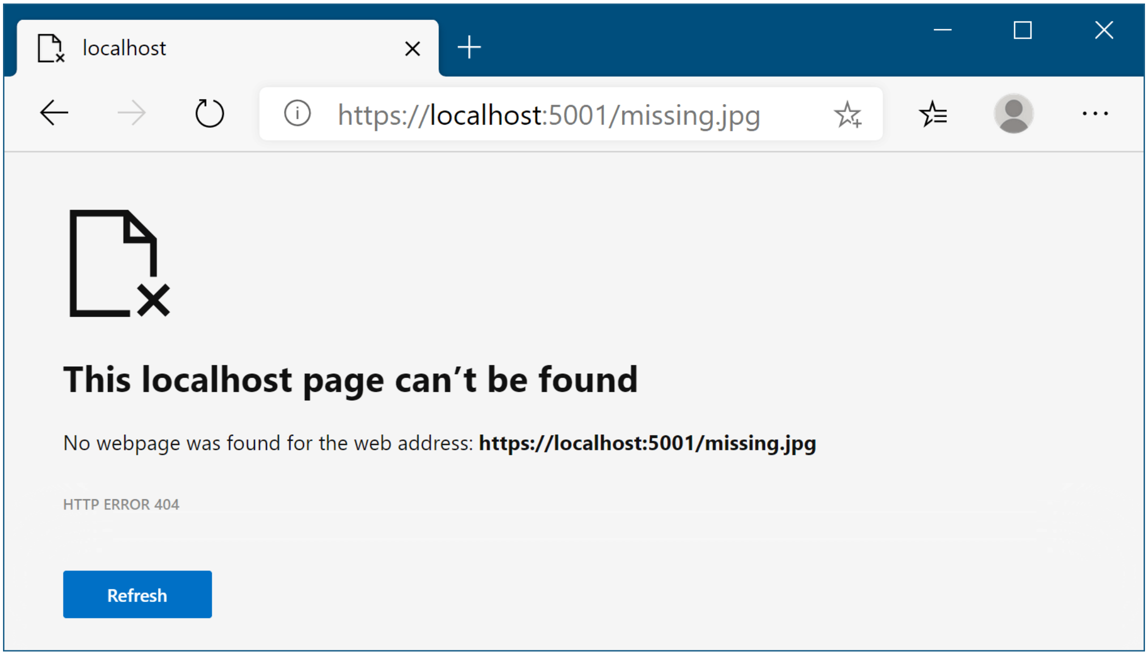
`ExceptionHandlerMiddleware` and `DeveloperExceptionPageMiddleware` are great for catching exceptions in your application, but exceptions aren't the only sort of errors you'll encounter. In some cases, your middleware pipeline will return an HTTP error status code in the response. It's important to handle both exceptions and error status codes to provide a coherent user experience.

3.3.3 Handling other errors: StatusCodePagesMiddleware

Your application can return a wide range of HTTP status codes that indicate some sort of error state. You've already seen that a 500 "server error" is sent when an exception occurs and isn't handled and that a 404 "file not found" error is sent when a URL isn't handled by any middleware. 404 errors, in particular, are common, often occurring when a user enters an invalid URL.

> **TIP** As well as indicating a completely unhandled URL, 404 errors are often used to indicate that a specific requested object was not found. For example, a request for the details of a product with an ID of 23 might return a 404 if no such product exists.

Without handling these status codes, users will see a generic error page, such as in figure 3.18, which may leave many confused and thinking your application is broken. A better approach would be to handle these error codes and return an error page that's in keeping with the rest of your application or, at the very least, doesn't make your application look broken.



This localhost page can’t be found

No webpage was found for the web address: **https://localhost:5001/missing.jpg**

HTTP ERROR 404

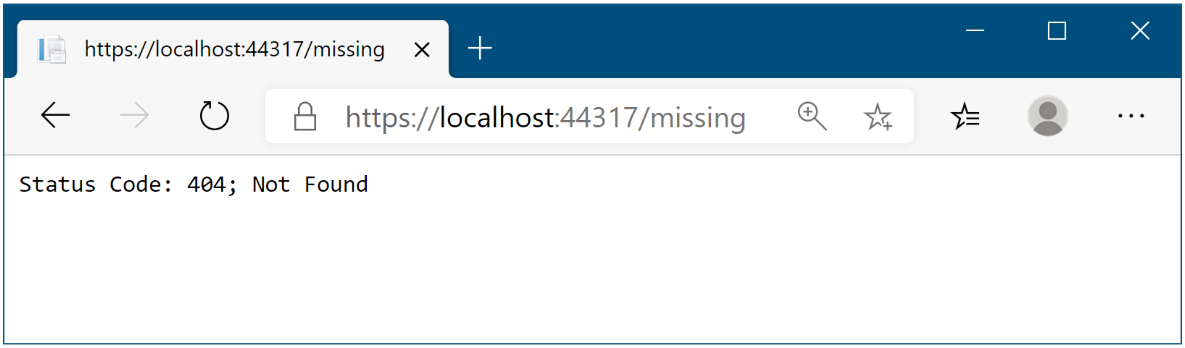
Figure 3.18 A generic browser error page. If the middleware pipeline can’t handle a request, it will return a 404 error to the user. The message is of limited usefulness to users and may leave many confused or thinking your web application is broken.

Microsoft provides `StatusCodePagesMiddleware` for handling this use case. As with all error handling middleware, you should add it early in your middleware pipeline, as it will only handle errors generated by later middleware components.

You can use the middleware a number of different ways in your application. The simplest approach is to add the middleware to your pipeline without any additional configuration, using

```
app.UseStatusCodePages();
```

With this method, the middleware will intercept any response that has an HTTP Status code that starts with 4xx or 5xx and has no response body. For the simplest case, where you don’t provide any additional configuration, the middleware will add a plain text response body, indicating the type and name of the response, as shown in figure 3.19. This is arguably worse than the default message at this point, but it is a starting point for providing a more consistent experience to users!


https://localhost:44317/missing
https://localhost:44317/missing

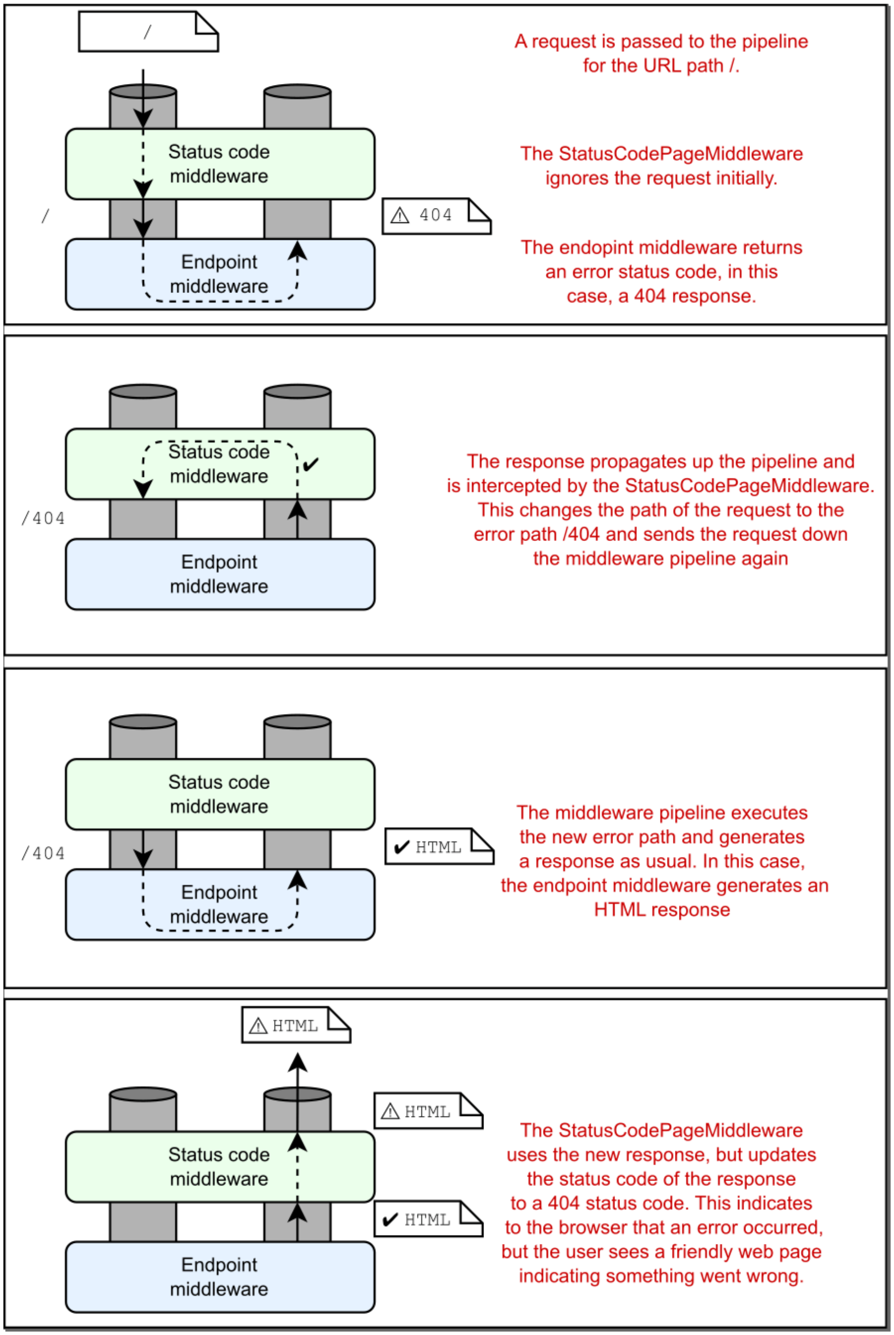
```
Status Code: 404; Not Found
```

Figure 3.19 Status code error pages for a 404 error. You generally won't use this version of the middleware in production as it doesn't provide a great user experience, but it demonstrates that the error codes are being correctly intercepted.

A more typical approach to using `StatusCodePageMiddleware` in production is to re-execute the pipeline when an error is captured, using a similar technique to the `ExceptionHandlerMiddleware`. This allows you to have dynamic error pages that fit with the rest of your application. To use this technique, replace the call to `UseStatusCodePages` with the following extension method

```
app.UseStatusCodePagesWithReExecute("/{0}");
```

This extension method configures `StatusCodePageMiddleware` to re-execute the pipeline whenever a 4xx or 5xx response code is found, using the provided error handling path. This is similar to the way `ExceptionHandlerMiddleware` re-executes the pipeline, as shown in figure 3.20.



A request is passed to the pipeline for the URL path /.

The StatusCodePageMiddleware ignores the request initially.

The endopint middleware returns an error status code, in this case, a 404 response.

The response propagates up the pipeline and is intercepted by the StatusCodePageMiddleware. This changes the path of the request to the error path /404 and sends the request down the middleware pipeline again

The middleware pipeline executes the new error path and generates a response as usual. In this case, the endpoint middleware generates an HTML response

The StatusCodePageMiddleware uses the new response, but updates the status code of the response to a 404 status code. This indicates to the browser that an error occurred, but the user sees a friendly web page indicating something went wrong.

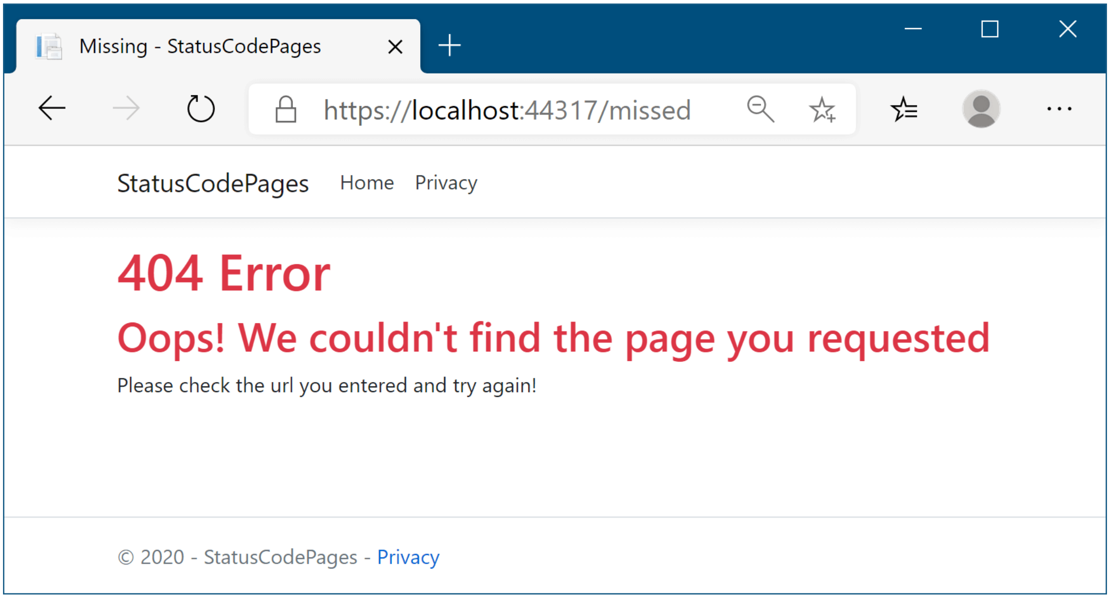
Figure 3.20 `StatusCodePagesMiddleware` re-executing the pipeline to generate an HTML body for a 404 response. A request to the / path returns a 404 response, which is handled by the status code middleware. The pipeline is re-executed using the / 404 path to generate the HTML response.

Note that the error handling path `"/{0}"` contains a format string token, `{0}`. When the path is re-executed, the middleware will replace this token with the status code number. For example, a 404 error would re-execute the `/404` path. The handler for the path (typically a Razor Page) has access to the status code and can optionally tailor the response, depending on the status code. You can choose any error handling path, as long as your application knows how to handle it.

NOTE You'll learn about how routing maps request paths to Razor Pages in chapter 5.

With this approach in place, you can create different error pages for different error codes, such as the 404-specific error page shown in figure 3.21. This technique ensures your error pages are consistent with the rest of your application, including any dynamically generated content, while also allowing you to tailor the message for common errors.

WARNING As before, when re-executing the pipeline, you must be careful your error handling path doesn't generate any errors.



404 Error

Oops! We couldn't find the page you requested

Please check the url you entered and try again!

© 2020 - StatusCodePages - Privacy

Figure 3.21 An error status code page for a missing file. When an error code is detected (in this case, a 404 error), the middleware pipeline is re-executed to generate the response. This allows dynamic portions of your web page to remain consistent on error pages.

You can use the StatusCodePagesMiddleware in combination with other exception handling middleware by adding both to the pipeline. StatusCodePagesMiddleware will only modify the response if no response body has been written. So if another component, for example the ExceptionHandlerMiddleware, returns a message body along with an error code, it won't be modified.

> **NOTE** StatusCodePageMiddleware **has additional overloads that let you execute custom middleware when an error occurs, instead of re-executing a Razor Pages path.**

Error handling is essential when developing any web application; errors happen, and you need to handle them gracefully. But depending on your application, you may not always want your error handling middleware to generate HTML pages.

3.3.4 Error handling middleware and Web APIs

ASP.NET Core isn't only great for creating user-facing web applications, it's also great for creating HTTP services that can be accessed either from another server application, from a mobile app, or from a user's browser when running a client-side single-page application. In all these cases, you probably won't be returning HTML to the client, but rather XML or JSON.

In that situation, if an error occurs, you probably don't want to be sending back a big HTML page saying, "Oops, something went wrong." Returning an HTML page to an application that's expecting JSON could easily break it unexpectedly. Instead, the HTTP 500 status code and a JSON body describing the error is more useful to a consuming application. Luckily, ASP.NET Core allows you to do exactly this when you create Web API controllers.

> **NOTE I'll discuss MVC and Web API controllers in chapter 4. I discuss Web APIs and handling errors in detail in chapter 9.**

That brings us to the end of middleware in ASP.NET Core for now. You've seen how to use and compose middleware to form a pipeline, as well as how to handle errors in your application. This will get you a long way when you start building your first ASP.NET Core applications. Later, you'll learn how to build your own custom middleware, as well as how to perform complex operations on the middleware pipeline, such as forking it in response to specific requests.

In the next chapter, you'll look in more depth at Razor Pages, and how they can be used to build websites. You'll also learn about the MVC design pattern, its relationship with Razor Pages in ASP.NET Core, and when to choose one approach over the other.

3.4 Summary

- Middleware has a similar role to HTTP modules and handlers in ASP.NET but is more easily reasoned about.

- Middleware is composed in a pipeline, with the output of one middleware passing to the input of the next.
- The middleware pipeline is two-way: requests pass through each middleware on the way in and responses pass back through in the reverse order on the way out.
- Middleware can short-circuit the pipeline by handling a request and returning a response, or it can pass the request on to the next middleware in the pipeline.
- Middleware can modify a request by adding data to, or changing, the `HttpContext` object.
- If an earlier middleware short-circuits the pipeline, not all middleware will execute for all requests.
- If a request isn't handled, the middleware pipeline will return a 404 status code.
- The order in which middleware is added to `IApplicationBuilder` defines the order in which middleware will execute in the pipeline.
- The middleware pipeline can be re-executed, as long as a response's headers haven't been sent.
- When added to a middleware pipeline, `StaticFileMiddleware` will serve any requested files found in the wwwroot folder of your application.
- `DeveloperExceptionPageMiddleware` provides a lot of information about errors when developing an application but should never be used in production.
- `ExceptionHandlerMiddleware` lets you provide user-friendly custom error handling messages when an exception occurs in the pipeline.
- `StatusCodePagesMiddleware` lets you provide user-friendly custom error handling messages when the pipeline returns a raw error response status code.
- Microsoft provides some common middleware and there are many third-party options available on NuGet and GitHub.

4
Creating a web site with Razor Pages

This chapter covers

- **Introducing Razor Pages and the Model-View-Controller (MVC) design pattern**
- **Using Razor Pages in ASP.NET Core**
- **Choosing between Razor Pages and MVC controllers**
- **Controlling application flow using action results**

In chapter 3, you learned about the middleware pipeline, which defines how an ASP.NET Core application responds to a request. Each piece of middleware can modify or handle an incoming request, before passing the request to the next middleware in the pipeline.

In ASP.NET Core web applications, your middleware pipeline will normally include the `EndpointMiddleware`. This is typically where you write the bulk of your application logic, by calling various other classes in your app. It also serves as the main entry point for users to interact with your app. It typically takes one of three forms:

- *An HTML web application, designed for direct use by users.* If the application is consumed directly by users, as in a traditional web application, then Razor Pages is responsible for generating the web pages that the user interacts with. It handles requests for URLs, it receives data posted using forms, and it generates the HTML that users use to view and navigate your app.
- *An API designed for consumption by another machine or in code.* The other main possibility for a web application is to serve as an API to backend server processes, to a mobile app, or to a client framework for building single page applications (SPAs). In

this case, your application serves data in machine-readable formats such as JSON or XML, instead of the human-focused HTML output.

- *Both an HTML web application, and an API.* It is also possible to have applications that serve both needs! This can let you cater to a wider range of clients, while sharing logic in your application

In this chapter, you'll learn how ASP.NET Core uses Razor Pages to handle the first of these options, creating server-side rendered HTML pages. You'll start by looking at the Model-View-Controller (MVC) design pattern to see the benefits that can be achieved through its use and learn why it's been adopted by so many web frameworks as a model for building maintainable applications.

Next, you'll learn how the MVC design pattern applies to ASP.NET Core. The MVC pattern is a broad concept that can be applied in a variety of situations, but the use case in ASP.NET Core is specifically as a UI abstraction. You'll see how Razor Pages implements the MVC design pattern, and how it builds on top of the ASP.NET Core MVC framework, comparing the two approaches.

Next, you'll see how to add Razor Pages to an existing application, and how to create your first Razor Pages. You'll learn how to define page handlers to execute when your application receives a request and how to generate a result that can be used to create an HTTP response to return.

I won't cover how to create Web APIs in this chapter. Web APIs still use the ASP.NET Core MVC framework, but they're used in a slightly different way to Razor Pages. Instead of returning web pages that are directly displayed on a user's browser, they return data formatted for consumption in code. Web APIs are often used for providing data to mobile and web applications, or to other server applications. But they still follow the same general MVC pattern. You'll see how to create a Web API in chapter 9.

> **NOTE** This chapter is the first of several on Razor Pages and MVC in ASP.NET Core. As I've already mentioned, these frameworks are often responsible for handling all the business logic and UI code for your application, so, perhaps unsurprisingly, they're large and somewhat complicated. The next five chapters all deal with a different aspect of the MVC pattern that make up the MVC and Razor Pages frameworks.

In this chapter, I'll try to prepare you for each of the upcoming topics, but you may find that some of the behavior feels a bit like magic at this stage. Try not to become too concerned with exactly how all the pieces tie together; focus on the specific concepts being addressed. It should all become clear as we cover the associated details in the remainder of this first part of the book.

4.1 An introduction to Razor Pages

The Razor Pages programming model was introduced in ASP.NET Core 2.0 as a way to build server-side rendered "page-based" web sites. It builds on top of the ASP.NET Core infrastructure to provide a streamlined experience, using conventions where possible to reduce the amount of boilerplate code and configuration required.

> **DEFINITION A *page-based* web site is one in which the user browses between multiple pages, enters data into forms, and generally consumes content. This contrasts with applications like games or single page applications (SPAs), which are heavily interactive on the client-side.**

You've already seen a very basic example of a Razor Page in chapter 2. In this section we'll start by looking at a slightly more complex Razor Page, to better understand the overall design of Razor Pages. I cover:

- An example of a typical Razor Page
- The MVC design pattern and how it applies to Razor Pages
- How to add Razor Pages to your application

At the end of this section you should have a good understanding of the overall design behind Razor Pages, and how they relate to the MVC pattern.

4.1.1 Exploring a typical Razor Page

In chapter 2 we looked at a very simple Razor Page. It didn't contain any logic, and instead just rendered the associated Razor view. This pattern may be common if you're building a content-heavy marketing website, for example, but more commonly your Razor Pages will contain some logic, load data from a database, or use forms to allow users to submit information, for example.

To give more of a flavor of how typical Razor Pages work, in this section we look briefly at a slightly more complex Razor Page. This page is taken from a to-do list application and is used to display all the to-do items for a given category. We're not focusing on the HTML generation at this point, so the listing below shows only the Page Model code-behind for the Razor Page.

Listing 4.1 A Razor Page for viewing all to-do items in a given category

```
public class CategoryModel : PageModel
{
    private readonly ToDoService _service;                  #A
    public CategoryModel(ToDoService service)               #A
    {
        _service = service;
    }

    public ActionResult OnGet(string category)              #B
    {
        Items = _service.GetItemsForCategory(category);     #C
        return Page();                                      #D
    }

    public List<ToDoListModel> Items { get; set; }          #E
}
```

#A The ToDoService is provided in the model constructor using dependency injection.
#B OnGet handler takes a parameter, category.
#C The handler calls out to the ToDoService to retrieve data and sets the Items property
#D Returns a PageResult indicating the Razor view should be rendered
#E The Razor View can access the Items property when it is rendered.

This example is still relatively simple, but it demonstrates a variety of features compared to the basic example from chapter 2:

- The page handler, `OnGet`, accepts a method parameter, `category`. This parameter is automatically populated by the Razor Page infrastructure using values from the incoming request, in a process called "model binding". I discuss model in detail in chapter 6.
- The handler doesn't interact with the database directly. Instead, it uses the `category` value provided to interact with the `ToDoService`, which is injected as a constructor argument using dependency injection.
- The handler returns `Page()` at the end of the method to indicate the associated Razor view should be rendered. The return statement is actually optional in this case; by convention, if the page handler is a `void` method, the Razor view will still be rendered, behaving as if you had called `return Page()` at the end of the method.
- The Razor View has access to the `CategoryModel` instance, so it can access the `Items` property that is set by the handler. It uses these items to build the HTML that is ultimately sent to the user.

The pattern of interactions in the Razor Page of listing 4.1 shows a common pattern. The page handler is the central controller for the Razor Page. It receives an input from the user (the `category` method parameter), calls out to the "brains" of the application (the `ToDoService`) and passes data (by exposing the `Items` property) to the Razor view which generates the HTML response. If you squint, this looks like the Model-View-Controller (MVC) design pattern.

Depending on your background in software development, you may have previously come across the MVC pattern in some form. In web development, MVC is a common paradigm and is used in frameworks such as Django, Rails, and Spring MVC. But as it's such a broad concept, you can find MVC in everything from mobile apps to rich-client desktop applications. Hopefully that's indicative of the benefits the pattern can bring if used correctly! In the next section I look at the MVC pattern in general and how it's used by ASP.NET Core.

4.1.2 The MVC design pattern

The MVC design pattern is a common pattern for designing apps that have UIs. The original MVC pattern has many different interpretations, each of which focuses on a slightly different aspect of the pattern. For example, the original MVC design pattern was specified with rich-client graphical user interface (GUI) apps in mind, rather than web applications, and so uses terminology and paradigms associated with a GUI environment. Fundamentally, though, the pattern aims to separate the management and manipulation of data from its visual representation.

Before I dive too far into the design pattern itself, let's consider a typical request. Imagine a user of your application requests the Razor Page from the previous section that displays a to-do list category. Figure 4.1 shows how a Razor Page handles different aspects of a request, all of which combine to generate the final response.

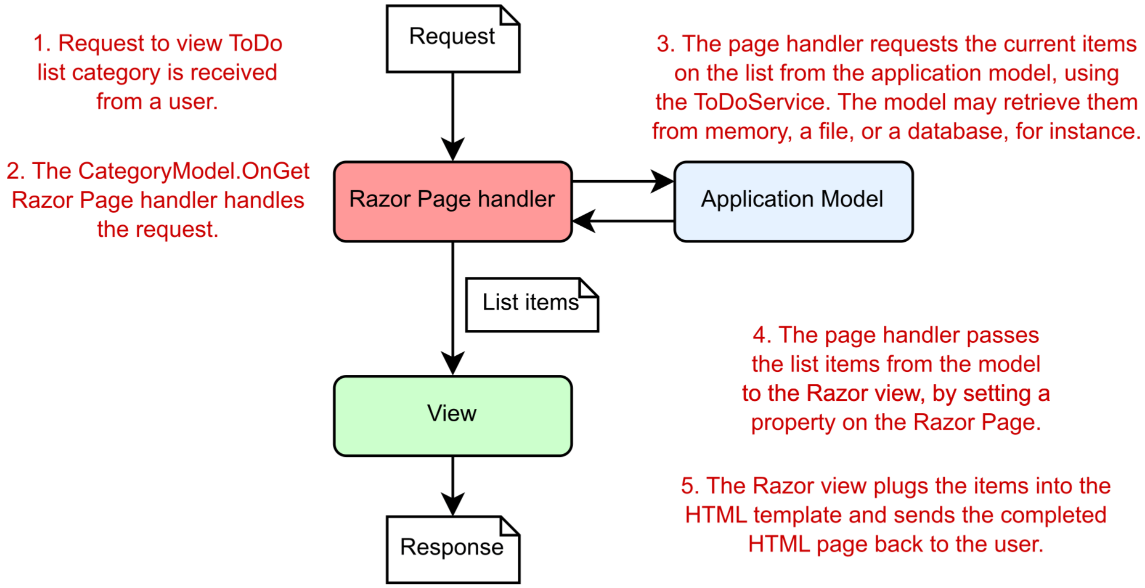

1. Request to view ToDo list category is received from a user.
Request
2. The CategoryModel.OnGet Razor Page handler handles the request.
Razor Page handler
3. The page handler requests the current items on the list from the application model, using the ToDoService. The model may retrieve them from memory, a file, or a database, for instance.
Application Model
List items
4. The page handler passes the list items from the model to the Razor view, by setting a property on the Razor Page.
View
5. The Razor view plugs the items into the HTML template and sends the completed HTML page back to the user.
Response

Figure 4.1 Requesting a to-do list page for a Razor Pages application. A different "component" handles each aspect of the request.

In general, three "components" make up the MVC design pattern:

- *Model*—The data that needs to be displayed, the global state of the application. Accessed via the `ToDoService` in Listing 4.1.
- *View*—The template that displays the data provided by the model.
- *Controller*—Updates the model and provides the data for display to the view. This role is taken by the page handler in Razor Pages. This is the `OnGet` method in listing 4.1.

Each component in the MVC design pattern is responsible for a single aspect of the overall system that, when combined, can be used to generate a UI. The to-do list example considers MVC in terms of a web application using Razor Pages, but a request could also be equivalent to the click of a button in a desktop GUI application.

In general, the order of events when an application responds to a user interaction or request is as follows:

1. The controller (the Razor Page handler) receives the request.
2. Depending on the request, the controller either fetches the requested data from the application model using injected services, or it updates the data that makes up the model.
3. The controller selects a view to display and passes a representation of the model to it.
4. The view uses the data contained in the model to generate the UI.

When we describe MVC in this format, the controller (the Razor Page handler) serves as the entry point for the interaction. The user communicates with the controller to instigate an interaction. In web applications, this interaction takes the form of an HTTP request, so when a request to a URL is received, the controller handles it.

Depending on the nature of the request, the controller may take a variety of actions, but the key point is that the actions are undertaken using the application model. The model here contains all the business logic for the application, so it's able to provide requested data or perform actions.

> **NOTE** In this description of MVC, the model is considered to be a complex beast, containing all the logic for how to perform an action, as well as any internal state. The Razor Page `PageModel` class is **not** the model we're talking about! Unfortunately, as in all software development, naming things is hard!

Consider a request to view a product page for an e-commerce application, for example. The controller would receive the request and would know how to contact some product service that's part of the application model. This might fetch the details of the requested product from a database and return them to the controller.

Alternatively, imagine a controller receives a request to add a product to the user's shopping cart. The controller would receive the request, and most likely invoke a method on the model to request that the product be added. The model would then update its internal representation of the user's cart, by adding, for example, a new row to a database table holding the user's data.

> **TIP** You can think of each Razor Page handler as a mini controller focused on a single page. Every web request is another independent call to a controller, which orchestrates the response. Although there are many different controllers, the handlers all interact with the *same* application model.

After the model has been updated, the controller needs to decide what response to generate. One of the advantages of using the MVC design pattern is that the model representing the application's data is decoupled from the final representation of that data, called the view. The controller is responsible for deciding whether the response should generate an HTML view, whether it should send the user to a new page, or whether it should return an error page.

One of the advantages of the model being independent of the view is that it improves testability. UI code is classically hard to test, as it's dependent on the environment—anyone who has written UI tests simulating a user clicking buttons and typing in forms knows that it's typically fragile. By keeping the model independent of the view, you can ensure the model stays easily testable, without any dependencies on UI constructs. As the model often contains your application's business logic, this is clearly a good thing!

The view can use the data passed to it by the controller to generate the appropriate HTML response. The view is only responsible for generating the final representation of the data, it's not involved in any of the business logic.

This is all there is to the MVC design pattern in relation to web applications. Much of the confusion related to MVC seems to stem from slightly different uses of the term for slightly different frameworks and types of application. In the next section, I'll show how the ASP.NET Core framework uses the MVC pattern with Razor Pages, along with more examples of the pattern in action.

4.1.3 Applying the MVC design pattern to Razor Pages

In the previous section I discussed the MVC *pattern* as typically used in web applications, and as used by Razor Pages. But ASP.NET Core also includes a *framework* called ASP.NET Core MVC. This framework (unsurprisingly) very closely mirrors the MVC design pattern, using *controllers* and *action methods* in place of Razor Pages and page handlers. Razor Pages builds directly on top of the underlying ASP.NET Core MVC framework, using the MVC framework "under the hood" for their behavior.

If you prefer, you can avoid Razor Pages entirely, and work with the MVC framework directly in ASP.NET Core. This was the *only* option in early versions of ASP.NET Core and the previous version of ASP.NET.

TIP I look in greater depth at choosing between Razor Pages and the MVC framework in section 4.2.

In this section we look in greater depth at how the MVC design pattern applies to Razor Pages in ASP.NET Core. This will also help clarify the role of various features of Razor Pages.

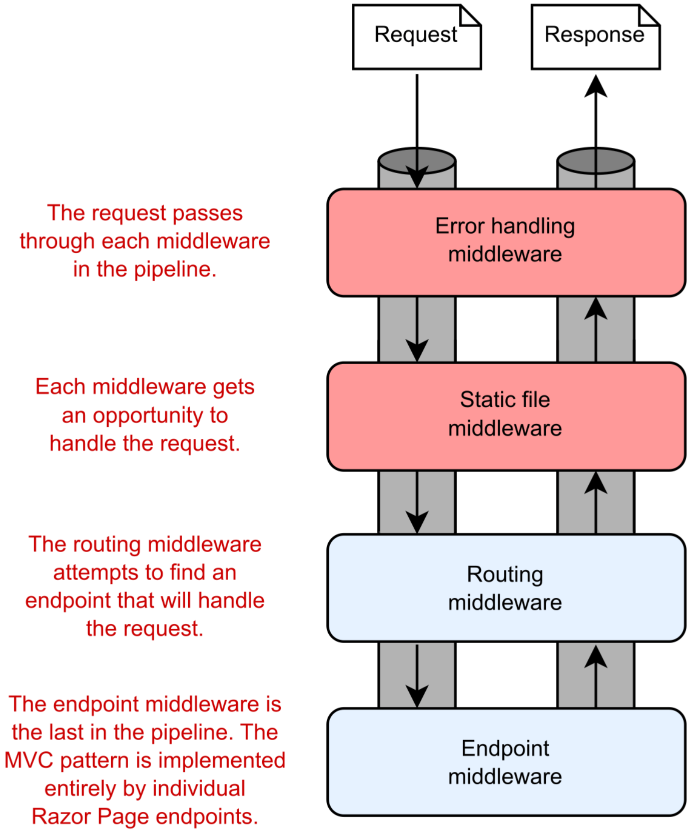
Do Razor Pages use MVC or MVVM?

Occasionally I've seen people describe Razor Pages as using the Model-View-View Model (MVVM) design pattern, rather than the MVC design pattern. Personally, I don't agree, but it's worth being aware of the differences.

MVVM is a UI pattern that is often used in mobile apps, desktop apps, and in some client-side frameworks. It differs from MVC in that there is a bi-directional interaction between the view and the view model. The view model tells the view what to display, but the view can also trigger changes directly on the view model. It's often used with two-way databinding where a view model is "bound" to a view.

Some people consider the Razor Pages `PageModel` to be filling this role, but I'm not convinced. Razor Pages definitely seems based on the MVC pattern to me (it's based on the ASP.NET Core MVC framework after all!) and doesn't have the same "two-way binding" that I would expect with MVVM.

As you've seen in previous chapters, ASP.NET Core implements Razor Page endpoints using a combination of the `RoutingMiddleware` and `EndpointMiddleware`, as shown in figure 4.2. Once a request has been processed by earlier middleware (and assuming none of them handle the request and short-circuit the pipeline), the routing middleware will select which Razor Page handler should be executed, and the Endpoint middleware executes the page handler.



The request passes through each middleware in the pipeline.

Each middleware gets an opportunity to handle the request.

The routing middleware attempts to find an endpoint that will handle the request.

The endpoint middleware is the last in the pipeline. The MVC pattern is implemented entirely by individual Razor Page endpoints.

Figure 4.2 The middleware pipeline for a typical ASP.NET Core application. The request is processed by each middleware in sequence. If the request reaches the routing middleware, the middleware selects an endpoint, such as a Razor Page, to execute. The endpoint middleware executes the selected endpoint.

Middleware often handles cross-cutting concerns or narrowly defined requests, such as requests for files. For requirements that fall outside of these functions, or that have many external dependencies, a more robust framework is required. Razor Pages (and/or ASP.NET Core MVC) can provide this framework, allowing interaction with your application's core business logic, and the generation of a UI. It handles everything from mapping the request to an appropriate controller to generating the HTML or API response.

In the traditional description of the MVC design pattern, there's only a single type of model, which holds all the non-UI data and behavior. The controller updates this model as appropriate and then passes it to the view, which uses it to generate a UI.

One of the problems when discussing MVC is the vague and ambiguous terms that it uses, such as "controller" and "model." Model, in particular, is such an overloaded term that it's often difficult to be sure exactly what it refers to—is it an object, a collection of objects, an abstract concept? Even ASP.NET Core uses the word "model" to describe several related, but different, components, as you'll see shortly.

DIRECTING A REQUEST TO A RAZOR PAGE AND BUILDING A BINDING MODEL

The first step when your app receives a request is routing the request to an appropriate Razor Page handler. Let's think about the category to-do list page again, from Listing 4.1. On this page, you're displaying a list of items that have a given category label. If you're looking at the list of items with a category of "Simple," you'd make a request to the `/category/Simple` URL.

Routing takes the headers and path of the request, `/category/Simple`, and maps it against a preregistered list of patterns. These patterns match a path to a single Razor Page and page handler. You'll learn more about routing in the next chapter.

> **TIP** I'm using the term Razor Page to refer to the combination of the Razor view and the `PageModel` that includes the page handler. Note that that `PageModel` class is *not* the "model" we're referring to when describing the MVC pattern. It fulfills other roles, as you will see later in this section.

Once a page handler is selected, the *binding model* (if applicable) is generated. This model is built based on the incoming request, the properties of the `PageModel` marked for binding, and the method parameters required by the page handler, as shown in figure 4.3. A binding model is normally one or more standard C# objects, with properties that map to the requested data. We'll look at binding models in detail in chapter 6.

> **DEFINITION** A *binding model* is one or more objects that act as a "container" for the data provided in a request that's required by a page handler.

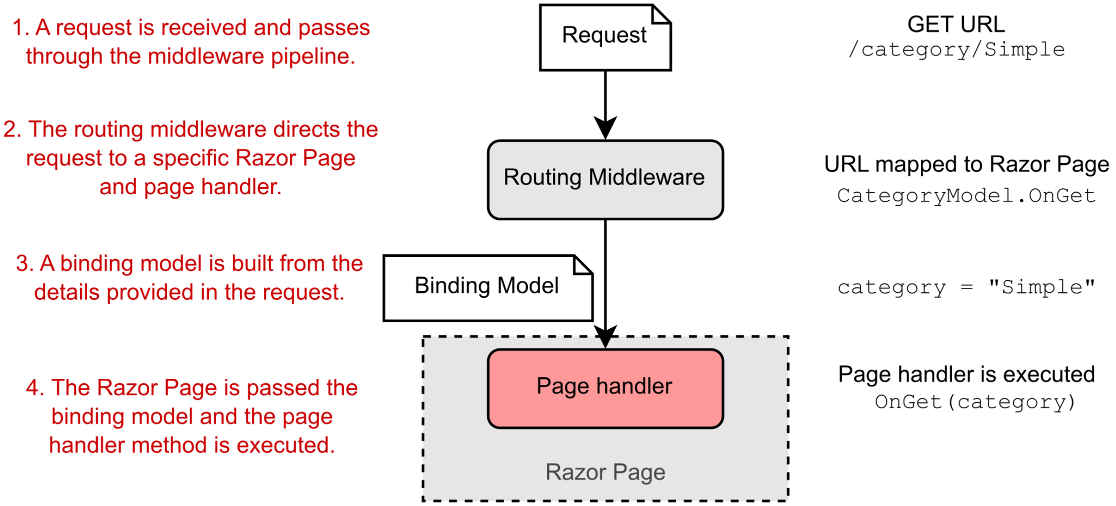

1. A request is received and passes through the middleware pipeline.
2. The routing middleware directs the request to a specific Razor Page and page handler.
3. A binding model is built from the details provided in the request.
4. The Razor Page is passed the binding model and the page handler method is executed.
Request
Routing Middleware
Binding Model
Page handler
Razor Page
GET URL
/category/Simple
URL mapped to Razor Page
CategoryModel.OnGet
category = "Simple"
Page handler is executed
OnGet(category)

Figure 4.3 Routing a request to a controller and building a binding model. A request to the `/category/Simple` URL results in the `CategoryModel.OnGet` page handler being executed, passing in a populated binding model, `category`.

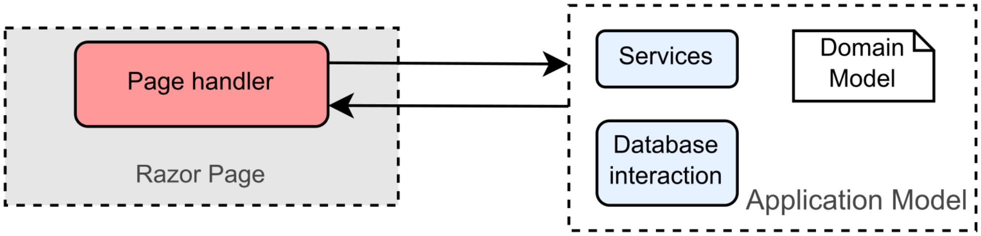
In this case, the binding model is a simple string, `category`, which is "bound" to the `"Simple"` value. This value is provided in the request URL's path. A more complex binding model could also have been used, where multiple properties were populated.

This binding model in this case corresponds to the method parameter of the `OnGet` page handler. An instance of the Razor Page is created using it's constructor, and the binding model is passed to the page handler when it executes, so it can be used to decide how to respond. For this example, the page handler uses it to decide which to-do items to display on the page.

EXECUTING A HANDLER USING THE APPLICATION MODEL

The role of the page handler as the controller in the MVC pattern is to *coordinate* the generation of a response to the request it's handling. That means it should perform only a limited number of actions. In particular, it should

- Validate that the data contained in the binding model provided is valid for the request.
- Invoke the appropriate actions on the application model using services.
- Select an appropriate response to generate based on the response from the application model.



1. The page handler uses the category provided in the binding model to determine which method to invoke in the application model.

2. The page handler method calls into services that make up the application model. This might use the domain model to determine whether to include completed ToDo items, for example.

3. The services load the details of the ToDO items from the database and return them back to the action method.

Figure 4.4 When executed, an action will invoke the appropriate methods in the application model.

Figure 4.4 shows the page handler invoking an appropriate method on the application model. Here, you can see that the application model is a somewhat abstract concept that encapsulates the remaining non-UI part of your application. It contains the *domain model*, a number of services, and the database interaction.

> **DEFINITION** The *domain model* encapsulates complex business logic in a series of classes that don't depend on any infrastructure and can be easily tested.

The page handler typically calls into a single point in the application model. In our example of viewing a to-do list category, the application model might use a variety of services to check whether the current user is allowed to view certain items, to search for items in the given category, to load the details from the database, or to load a picture associated with an item from a file.

Assuming the request is valid, the application model will return the required details to the page handler. It's then up to the page handler to choose a response to generate.

BUILDING HTML USING THE VIEW MODEL

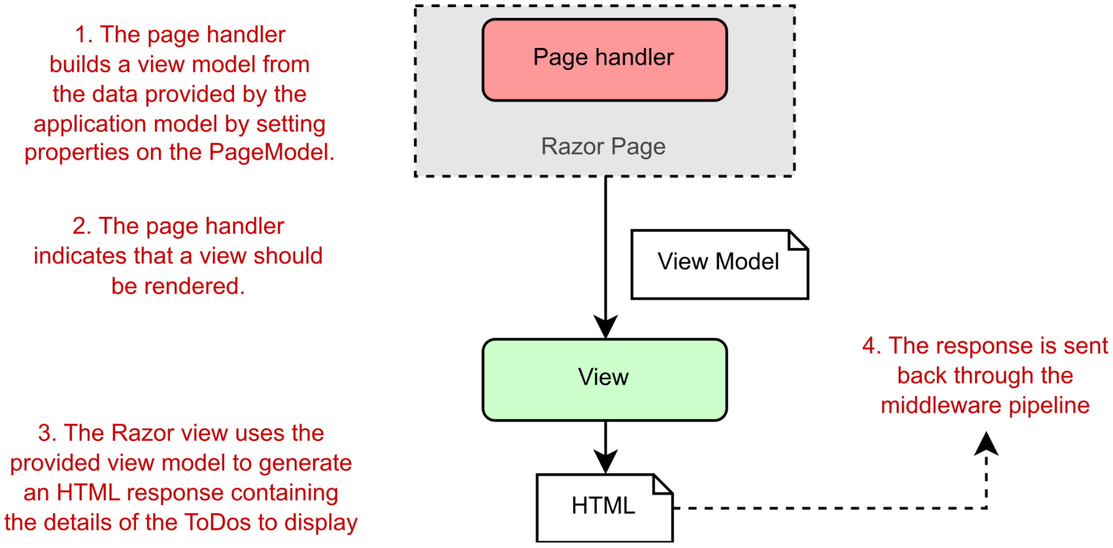
Once the page handler has called out to the application model that contains the application business logic, it's time to generate a response. A *view model* captures the details necessary for the view to generate a response.

> **DEFINITION** A *view model* is all the data required by the view to render a UI. It's typically some transformation of the data contained in the application model, plus extra information required to render the page, for example the page's title.

The term *view model* is used extensively in ASP.NET Core MVC, where it typically refers to a single object that is passed to the Razor view to render. However, with Razor Pages, the Razor view can access the Razor Page's *page model* class directly. Therefore, the Razor Page `PageModel` typically *acts* as the view model in Razor Pages, with the data required by the Razor view exposed via properties, as you saw previously in Listing 4.1.

> **NOTE** **Razor Pages use the `PageModel` class itself as the view model for the Razor view, by exposing the required data as properties.**

The Razor view uses the data exposed in the page model to generate the final HTML response. Finally, this is sent back through the middleware pipeline and out to the user's browser, as shown in figure 4.5.



1. The page handler builds a view model from the data provided by the application model by setting properties on the PageModel.

2. The page handler indicates that a view should be rendered.

3. The Razor view uses the provided view model to generate an HTML response containing the details of the ToDos to display

4. The response is sent back through the middleware pipeline

Figure 4.5 The page handler builds a view model by setting properties on the `PageModel`. It's the view that generates the response.

It's important to note that although the page handler selects *whether* to execute the view, and the data to use, it doesn't control *what HTML is generated*. It's the view itself that decides what the content of the response will be.

PUTTING IT ALL TOGETHER: A COMPLETE RAZOR PAGE REQUEST

Now that you've seen each of the steps that goes into handling a request in ASP.NET Core using Razor Pages, let's put it all together from request to response. Figure 4.6 shows how the steps combine to handle the request to display the list of to-do items for the "Simple" category. The traditional MVC pattern is still visible in Razor Pages, made up of the page handler (controller), the view, and the application model.

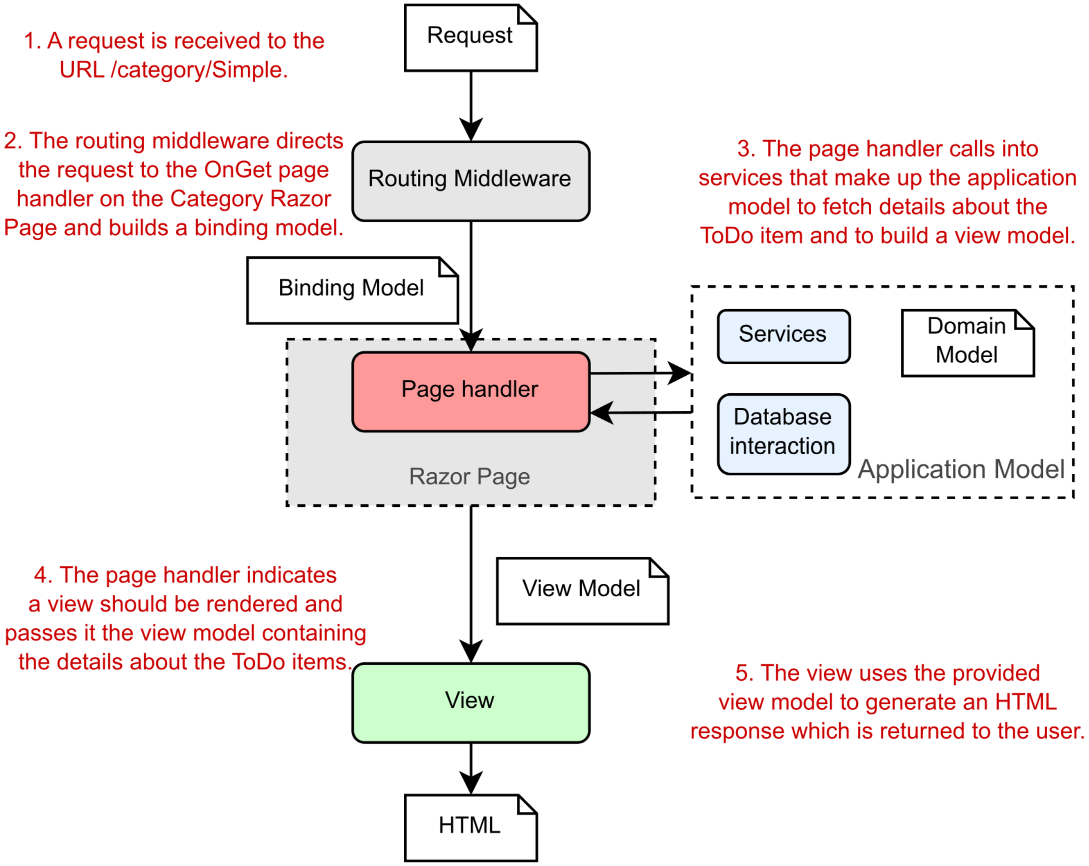

1. A request is received to the URL /category/Simple.
Request
2. The routing middleware directs the request to the OnGet page handler on the Category Razor Page and builds a binding model.
Routing Middleware
3. The page handler calls into services that make up the application model to fetch details about the ToDo item and to build a view model.
Binding Model
Services
Domain Model
Page handler
Database interaction
Razor Page
Application Model
4. The page handler indicates a view should be rendered and passes it the view model containing the details about the ToDo items.
View Model
View
5. The view uses the provided view model to generate an HTML response which is returned to the user.
HTML

Figure 4.6 A complete Razor Pages request for the list of to-dos in the "Simple" category.

By now, you might be thinking this whole process seems rather convoluted—so many steps to display some HTML! Why not allow the application model to create the view directly, rather than having to go on a dance back and forth with the page handler method?

The key benefit throughout this process is the *separation of concerns*:

- The view is responsible only for taking some data and generating HTML.
- The application model is responsible only for executing the required business logic.

- The page handler (controller) is responsible only for validating the incoming request and selecting the appropriate view to display, based on the output of the application model.

By having clearly defined boundaries, it's easier to update and test each of the components without depending on any of the others. If your UI logic changes, you won't necessarily have to modify any of your business logic classes, so you're less likely to introduce errors in unexpected places.

The dangers of tight coupling

Generally speaking, it's a good idea to reduce coupling between logically separate parts of your application as much as possible. This makes it easier to update your application without causing adverse effects or requiring modifications in seemingly unrelated areas. Applying the MVC pattern is one way to help with this goal.

As an example of when coupling rears its head, I remember a case a few years ago when I was working on a small web app. In our haste, we had not properly decoupled our business logic from our HTML generation code, but initially there were no obvious problems—the code worked, so we shipped it!

A few months later, someone new started working on the app, and immediately "helped" by renaming an innocuous spelling error in a class in the business layer. Unfortunately, the names of those classes had been used to generate our HTML code, so renaming the class caused the whole website to break in users' browsers! Suffice it to say, we made a concerted effort to apply the MVC pattern after that, and ensure we had a proper separation of concerns.

The examples shown in this chapter demonstrate the bulk of the Razor Pages functionality. It has additional features, such as the filter pipeline, that I'll cover later (chapter 13), and I'll discuss binding models in greater depth in chapter 6, but the overall behavior of the system is the same.

Similarly, in chapter 9, I'll discuss how the MVC design pattern applies when you're generating machine-readable responses using Web API controllers. The process is, for all intents and purposes, identical, apart from the final result generated.

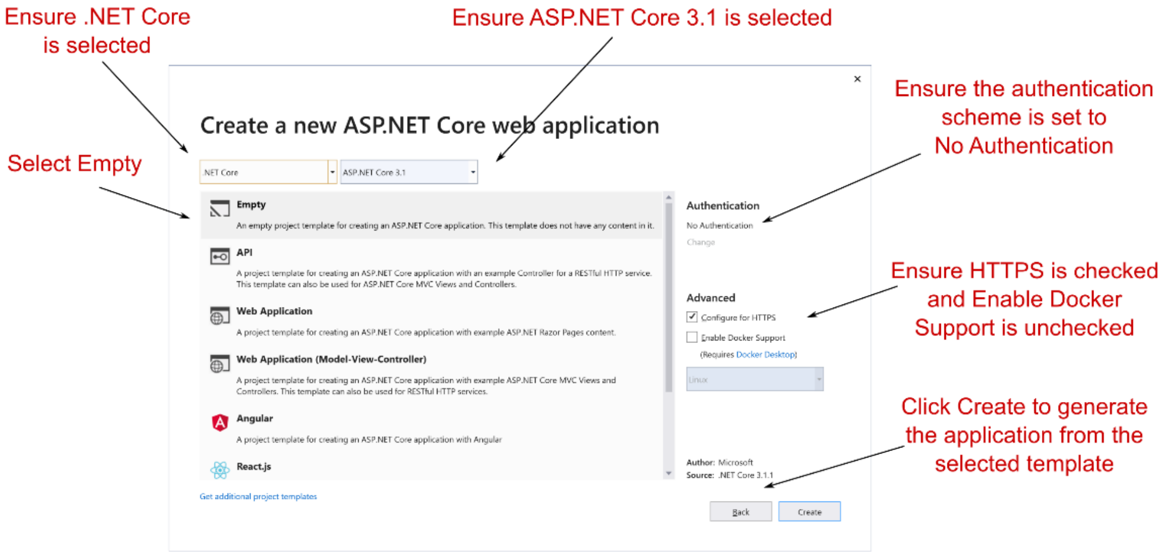
In the next section, you'll see how to add Razor Pages to your application. Some templates in Visual Studio and the .NET CLI will include Razor Pages by default, but you'll see how to add it to an existing application and explore the various options available.

4.1.4 Adding Razor Pages to your application

The MVC infrastructure, whether used by Razor Pages or MVC/API controllers, is a foundational aspect of all but the simplest ASP.NET Core applications, so virtually all templates include it configured by default in some way. But to make sure you're comfortable with adding Razor Pages to an existing project, I'll show how to start with a basic empty application and add Razor Pages to it from scratch.

The result of your efforts won't be exciting yet. We'll display "Hello World" on a web page, but it'll show how simple it is to convert an ASP.NET Core application to use Razor Pages. It also emphasizes the pluggable nature of ASP.NET Core—if you don't need the functionality provided by Razor Pages, then you don’t have to use it. Here’s how you add Razor Pages to your application:

1. In Visual Studio 2019, choose File > New > Project or choose Create a New Project from the splash screen
2. From the list of templates, choose ASP.NET Core Web Application, ensuring you select the C# language template.
3. On the next screen, enter a project name, Location, and a solution name, and click Create.
4. On the following screen create a basic template without MVC or Razor Pages by selecting the Empty Project template in Visual Studio, as shown in figure 4.8. You can create a similar empty project using the .NET CLI with the `dotnet new web` command.



Create a new ASP.NET Core web application

.NET Core | ASP.NET Core 3.1

Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Web Application
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

Web Application (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Angular
A project template for creating an ASP.NET Core application with Angular

React.js

Get additional project templates

Authentication
No Authentication
Change

Advanced
☑ Configure for HTTPS
□ Enable Docker Support
(Requires Docker Desktop)
Linux

Author: Microsoft
Source: .NET Core 3.1.1

Ensure .NET Core is selected

Ensure ASP.NET Core 3.1 is selected

Select Empty

Ensure the authentication scheme is set to No Authentication

Ensure HTTPS is checked and Enable Docker Support is unchecked

Click Create to generate the application from the selected template

Figure 4.8 Creating an empty ASP.NET Core template. The empty template will create a simple ASP.NET Core application that contains a small middleware pipeline without Razor Pages.

5. Add the necessary Razor Page services (in bold) in your Startup.cs file’s `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
}
```

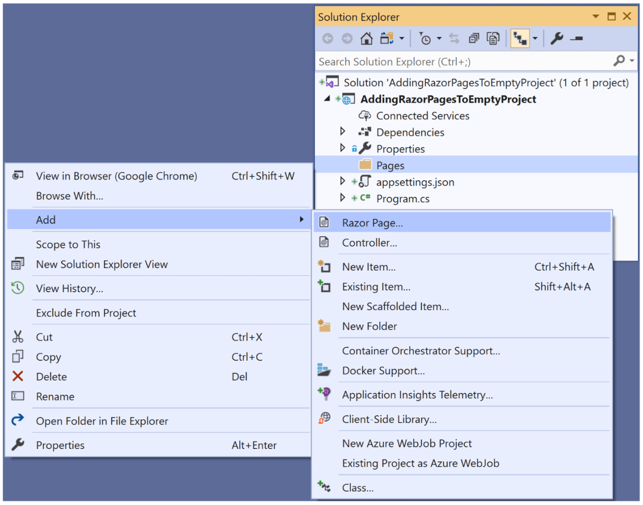
6. Replace the existing basic endpoint configured in the `EndpointMiddleware` at the end of your middleware pipeline with the `MapRazorPages()` extension method (in bold). For simplicity, also remove the existing error handler middleware from the `Configure` method of Startup.cs for now:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

7. Right-click your project in Solution Explorer and choose Add > New Folder to add a new folder to the root of your project. Name the new folder Pages.

   You have now configured your project to use Razor Pages, but you don't have any pages yet. The following steps add a new Razor Page to your application. You can create a similar Razor Page using the .NET CLI by running `dotnet new page -n Index -o Pages/` from the project directory.

8. Right-click the new pages folder and choose Add > Razor Page, as shown in figure 4.9.



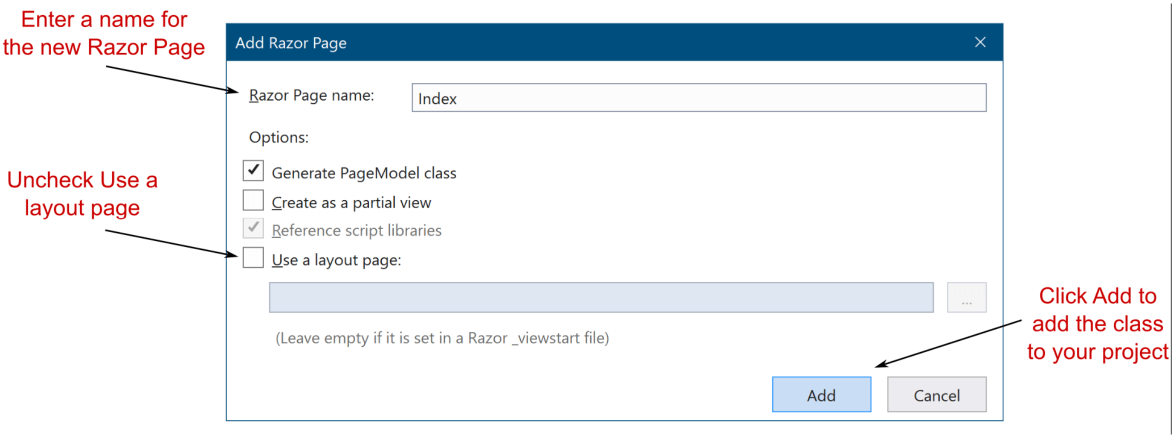
1. Right click on the Pages folder to bring up the context menu

2. Click Add to open the Add submenu

3. Choose Razor Page to add a Razor Page to your project

Figure 4.9 Adding a new Razor Page to your project

9. On the following page select Razor Page and click Add. In the following dialog box, name your page Index, uncheck the Use a layout page option, and click Add, as shown in figure 4.10. Leave all other checkboxes with their default values.



Enter a name for the new Razor Page

Uncheck Use a layout page

Click Add to add the class to your project

Add Razor Page

Razor Page name: Index

Options:

☑ Generate PageModel class

□ Create as a partial view

☑ Reference script libraries

□ Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

Figure 4.10 Creating a new Razor Page using the Add Razor Page dialog box

10. After Visual Studio has finished generating the file, open the Index.chtml file, and update the HTML to say `Hello World` by adding an `<h1>` element inside the `<body>` element (in bold):

```
@page
@model AddingRazorPagesToEmptyProject.IndexModel
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <h1>Hello World!</h1>
</body>
</html>
```

Once you’ve completed all these steps, you should be able to restore, build, and run your application.

> **NOTE** **You can run your project by pressing F5 from within Visual Studio (or by calling** `dotnet run` **at the command line from the project folder). This will restore any referenced NuGet packages, build your project,**

and start your application. Visual Studio will automatically open a browser window to access your application's homepage.

When you make a request to the root "/" path, the application invokes the `OnGet` handler on the `IndexModel` due to the conventional way routing works for Razor Pages based on the file name. Don't worry about this for now; we'll go into it in detail in the next chapter.

The `OnGet` handler is a `void` method, which causes the Razor Page to render the associated Razor view and send it to the user's browser in the response.

Razor Pages rely on a number of internal services to perform their functions, which must be registered during application startup. This is achieved with the call to `AddRazorPages` in the `ConfigureServices` method of Startup.cs. Without this, you'll get exceptions when your app starts, reminding you that the call is required.

The call to `MapRazorPages` in `Configure` registers the Razor Page endpoints with the endpoint middleware. As part of this call, the routes that are used to map URL paths to a specific Razor Page handler are registered automatically.

NOTE I cover routing in detail in the next chapter.

The instructions in this section described how to add Razor Pages to your application, but it's not the only way to add HTML generation to your application. As I mentioned previously, Razor Pages builds on top of the ASP.NET Core MVC framework and shares many of the same concepts. In the next section, we take a brief look at MVC controllers, see how they compare to Razor Pages, and discuss when you should choose to use one approach over the other.

4.2 Razor Pages vs MVC in ASP.NET Core

In this book, I focus on Razor Pages, as that has become the recommended approach for building server-side rendered applications with ASP.NET Core. However, I also mentioned that Razor Pages uses the ASP.NET Core MVC framework behind the scenes, and that you can choose to use it directly if you wish. Additionally, if you're creating a Web API for working with mobile or client-side apps then you will almost certainly be using the MVC framework directly.

NOTE I look at how to build Web APIs in chapter 9.

So, what are the differences between Razor Pages and MVC, and when should you choose one or the other?

If you're new to ASP.NET Core, the answer is pretty simple—use Razor Pages for server-side rendered applications and use the MVC framework for building Web APIs. There are nuances I'll discuss in later sections, but that distinction will serve you very well initially.

If you're familiar with the previous version of ASP.NET or earlier versions of ASP.NET Core and are deciding whether to use Razor Pages, then this section should help you choose. Developers coming from those backgrounds often have misconceptions about Razor Pages initially (as I did!), incorrectly equating them to Web Forms and overlooking their underlying basis of the MVC framework.

Before we can get to comparisons though, we should take a brief look at the ASP.NET Core MVC framework itself. Understanding the similarities and differences between MVC and Razor Pages can be very useful, as and you'll likely find a use for MVC at some point, even if you use Razor Pages most of the time.

4.2.1 MVC controllers in ASP.NET Core

In section 4.1 we looked at the MVC design pattern, and how it applies to Razor Pages in ASP.NET Core. Perhaps unsurprisingly, you can use the ASP.NET Core MVC framework in almost exactly the same way. To demonstrate the difference between Razor Pages and MVC, we'll look at an MVC version of the Razor Page from Listing 4.1, which displays a list of to-do items for a given category.

Instead of a `PageModel` and page handler, MVC uses a concept of *controllers* and *action methods*. These are almost directly analogous to their Razor Page counterparts, as you can see in figure 4.7, which shows an MVC equivalent of figure 4.6. On the other hand, MVC controllers use explicit *view models* to pass data to a Razor view, rather than exposing the data as properties on itself (as Razor Pages do with Page Models).

> **DEFINITION** An *action* (or *action method*) is a method that runs in response to a request. An *MVC controller* is a class that contains a number of logically grouped action methods.

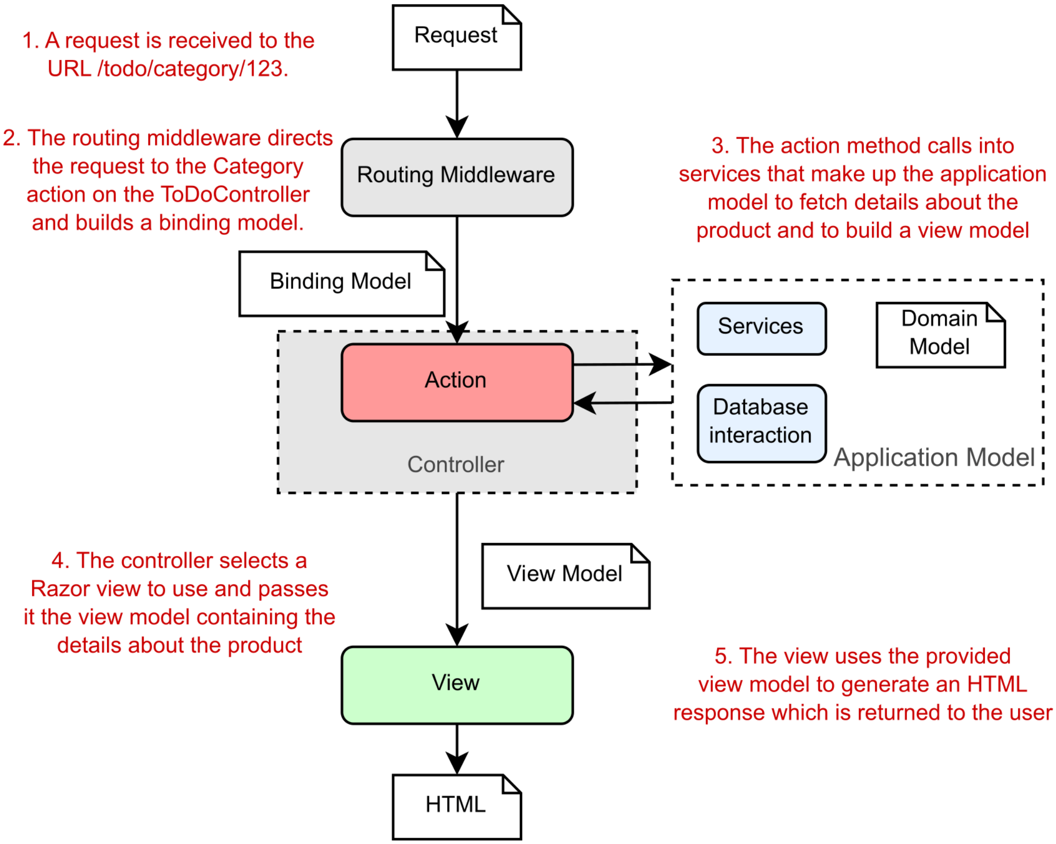

1. A request is received to the URL /todo/category/123.
Request
2. The routing middleware directs the request to the Category action on the ToDoController and builds a binding model.
Routing Middleware
3. The action method calls into services that make up the application model to fetch details about the product and to build a view model
Binding Model
Services
Domain Model
Action
Database interaction
Controller
Application Model
4. The controller selects a Razor view to use and passes it the view model containing the details about the product
View Model
View
5. The view uses the provided view model to generate an HTML response which is returned to the user
HTML

Figure 4.7 A complete MVC controller request for a category. The MVC controller pattern is almost identical to that of Razor Pages, shown in figure 4.6. The Controller is equivalent to a Razor Page, and the Action is equivalent to a page handler.

The listing below shows an example of how an MVC controller that provides the same functionality as the Razor Page in Listing 4.1 might look. In MVC, controllers are often used to aggregate similar actions together, so the controller in this case is called `ToDoController`, as it would typically contain additional action methods for working with to-do items, such as actions to view a specific item, or to create a new one.

Listing 4.2 An MVC controller for viewing all to-do items in a given category

```
public class ToDoController : Controller
{
    private readonly ToDoService _service;              #A
    public ToDoController(ToDoService service)          #A
    {
        _service = service;
    }

    public ActionResult Category(string id)              #B
    {
        var items = _service.GetItemsForCategory(id);    #C
        var viewModel = new CategoryViewModel(items);    #C

        return View(items);                              #D
    }

    public ActionResult Create(ToDoListModel model)      #E
    {                                                    #E
        // ...                                           #E
    }                                                    #E
}
```

#A The ToDoService is provided in the controller constructor using dependency injection.
#B The Category action method takes a parameter, id.
#C The action method calls out to the ToDoService to retrieve data and build a view model
#D Returns a ViewResult indicating the Razor view should be rendered, passing in the view model
#E MVC controllers often contain multiple action methods which respond to different requests

Aside from some naming differences, the `ToDoController` looks very similar to the Razor Page equivalent from Listing 4.1. Indeed, *architecturally*, Razor Pages and MVC are essentially equivalent, as they both use the MVC design pattern. The most obvious differences relate to *where the files are placed* in your project, as I discuss in the next section.

4.2.2 The benefits of Razor Pages

In the previous section I showed that the code for an MVC controller looks very similar to the code for a Razor Page `PageModel`. If that's the case, what benefit is there to using Razor Pages? In this section I discuss some of the pain points of MVC controllers and how Razor Pages attempts to address them.

Razor Pages are not web forms

A common argument I hear from existing ASP.NET developers against Razor Pages is "oh, they're just Web Forms". That sentiment misses the mark in many different ways, but it's common enough that it's worth addressing directly.

Web Forms was a web-programming model that was released as part of .NET Framework 1.0 in 2002. They attempted to provide a highly productive experience for developers moving from desktop development to the web for the first time.

Web Forms are much-maligned now, but their weakness only became apparent later. Web forms attempted to hide the complexities of the web away from you, to give you the "impression" of developing with a desktop app. That often resulted in apps that were slow, with lots of inter-dependencies, and were hard to maintain.

Web Forms provided a "page-based" programming model, which is why Razor Pages sometimes gets associated with them. However, as you've seen, Razor Pages is based on the MVC design pattern, and exposes the intrinsic features of the web without trying to hide them from you.

Razor Pages optimizes certain flows using conventions (some of which you've seen), but it's not trying to build a *stateful* application model over the top of a stateless web application, in the way that Web Forms did.

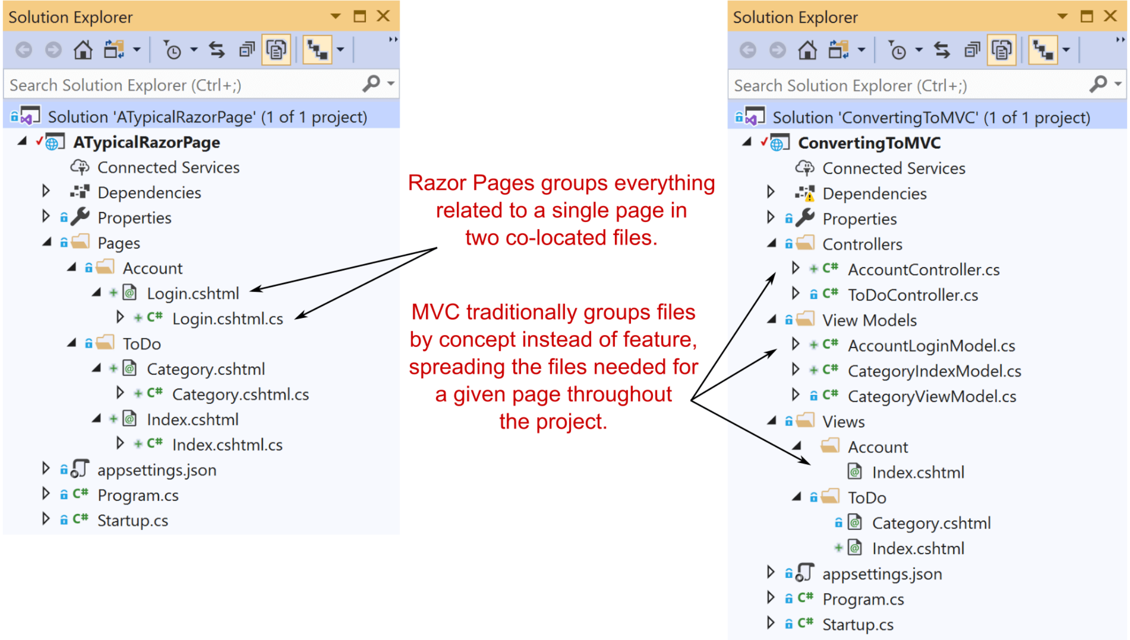
In MVC, a single controller can have multiple action methods. Each action handles a different request and generates a different response. The grouping of multiple actions in a controller is somewhat arbitrary, but is typically used to group actions related to a specific entity, to-do list items in this case. A more complete version of the `ToDoController` in Listing 4.2 might include action methods for listing all to-do items, for creating new items, and for deleting items, for example. Unfortunately, you can often find that your controllers become very large and bloated, with many dependencies.[15]

> **NOTE** You don't *have* to make your controllers very large like this, it's just a very common pattern. You could, for example, create a separate controller for every action instead.

Another pitfall of the MVC controllers are the way they're typically organized in your project. Most action methods in a controller will need an associated Razor view, and a view model for passing data to the view. The MVC approach traditionally groups classes by *type* (controller, view, view model), while the Razor Page approach groups by *function*—everything related to a specific page is co-located.

Figure 4.11 compares the file layout for a simple Razor Page project with the MVC equivalent. Using Razor Pages means much less scrolling up and down between the controller, views, and view model folders whenever you're working on a particular page. Everything you need is found in two files, the cshtml Razor view and the cshtml.cs `PageModel` file.

[15] Before moving to Razor Pages, the ASP.NET Core template that includes user login functionality contained two such controllers, each containing over 20 action methods, and over 500 lines of code!



Razor Pages groups everything related to a single page in two co-located files.

MVC traditionally groups files by concept instead of feature, spreading the files needed for a given page throughout the project.

Figure 4.11 Comparing the folder structure for an MVC project to the folder structure for a Razor Pages project.

There are additional differences between MVC and Razor Pages, which I'll highlight throughout the book, but this layout difference is really the biggest win. Razor Pages embraces the fact that you're building a page-based application and optimizes your workflow by keeping everything related to a single page together.

> **TIP** You can think of each Razor Page as a mini controller focused on a single page. Page handlers are functionally equivalent to MVC controller action methods.

This layout also has the benefit of making each page a separate class. This contrasts with the MVC approach of making each page an *action* on a given controller. Each Razor Page is cohesive for a particular *feature*, such as displaying a to-do item, for example. MVC controllers contain action methods which handle multiple different features for a more abstract *concept*, such as all the features related to to-do items.

Another important point is that Razor Pages doesn't lose any of the "separation-of-concerns" that MVC has. The view part of Razor Pages is still only concerned with rendering HTML and the handler is the "coordinator" that calls out to the application model. The only real difference is the lack of the explicit view model that you have in MVC, but it's perfectly possible to emulate this in Razor Pages if that's a deal breaker for you.

The benefits that Razor Pages bring are particularly noticeable when you have "content" websites, such as marketing websites, where you're mostly displaying static data, and there's no real logic. In that case, MVC adds complexity without any real benefits, as there's not really any logic in the controllers at all. Another great use case is when creating forms for users to submit data. Razor Pages is especially optimized for this scenario, as you'll see in later chapters.

Clearly, I'm a fan of Razor Pages, but that's not to say they're perfect for every situation. In the next section I discuss some of the cases when you might choose to use MVC controllers in your application. Bear in mind it's not an either-or choice—it's possible to use both MVC controllers and Razor Pages in the same application, and in many cases that may be the best option.

4.2.3 When to choose MVC controllers over Razor Pages

Razor Pages are great for building page-based server-side rendered applications. But not all applications fit that mold, and even some applications that *do* fall in that category might be best developed using MVC controllers instead of Razor Pages. Possible reasons include:

- *When you don't want to render views*. Razor Pages are best for page-based applications, where you're rending a view for the user. If you're building a Web API then you should use MVC controllers instead.
- *When you're converting an existing MVC application to ASP.NET Core*. If you already have an ASP.NET application that uses MVC, then it's probably not worth converting your existing MVC controllers to Razor Pages. It makes more sense to keep your existing code, and perhaps look at doing *new* development in the application with Razor Pages.
- *When you're doing a lot of partial page updates*. It's possible to use JavaScript in an application to avoid doing full page navigations, by only updating part of the page at a time. This approach, halfway between fully server-side rendered and a client-side application may be easier to achieve with MVC controllers than Razor Pages.

When not to use Razor Pages or MVC controllers

Typically, you'll use either Razor Pages or MVC controllers to write most of your application logic for an app. You'll use it to define the APIs and pages in your application, and to define how they interface with your business logic. Razor Pages and MVC provide an extensive framework (as you'll see over the next six chapters) that provide a great deal of functionality to help build your apps quickly and efficiently. But they're not suited to *every* app.

Providing so much functionality necessarily comes with a certain degree of performance overhead. For typical apps, the productivity gains from using MVC or Razor Pages strongly outweigh any performance impact. But if you're building small, lightweight apps for the cloud, then you might consider using custom middleware directly (see chapter 19). You might want to also consider *Microservices in .NET Core* by Christian Horsdal Gammelgaard (Manning, 2017).

Alternatively, if you're building an app with real-time functionality, you'll probably want to consider using WebSockets instead of traditional HTTP requests. ASP.NET Core SignalR can be used to add real-time functionality to your app by providing an abstraction over Web Sockets. SignalR also provides simple transport fallbacks and a remote procedure call (RPC) app model. For details, see the documentation at https://docs.microsoft.com/en-us/aspnet/core/signalr.

Another option introduced in ASP.NET Core 3.1 is Blazor. This framework allows you to build interactive client-side web applications by either leveraging the WebAssembly standard to run .NET code directly in your browser, or by using a stateful model with SignalR. See the documentation for details at https://docs.microsoft.com/en-us/aspnet/core/blazor/.

Hopefully by this point you're sold on Razor Pages and their overall design. So far, all the Razor Pages we've looked at have used a single page handler. In the next section we'll look in greater depth at page handlers, how to define them, how to invoke them, and how to use them to render Razor views.

4.3 Razor Pages and page handlers

In the first section of this chapter, I described the MVC design pattern and how it relates to ASP.NET Core. In the design pattern, the controller receives a request and is the entry point for UI generation. For Razor Pages, the entry point is a page handler that resides in a Razor Page's `PageModel`. A page handler is a method that runs in response to a request.

By default, the path of a Razor Page on disk controls the URL path that the Razor Page responds to. For example, a request to the URL `/products/list` corresponds to the Razor Page at the path pages/Products/List.cshtml. Razor Pages can contain any number of page handlers, but only one runs in response to a given request.

> **NOTE** You'll learn more about this process of selecting a Razor Page and handler, called *routing*, in the next chapter.

The responsibility of a page handler is generally threefold:

- Confirm the incoming request is valid.
- Invoke the appropriate business logic corresponding to the incoming request.
- Choose the appropriate *kind* of response to return.

A page handler doesn't need to perform all these actions, but at the very least it must choose the kind of response to return. Page Handlers typically return one of 3 things:

- A `PageResult` object. This causes the associated Razor view to generate an HTML response.
- Nothing (the handler returns `void` or `Task`). This is the same as the previous case, causing the Razor view to generate an HTML response.
- A `RedirectToPageResult`. This indicates the user should be redirected to a different page in your application.

These are the most commonly used results for Razor Pages, but I describe some additional options in section 4.3.2.

It's important to realize that an action method doesn't generate a response directly; it selects the *type* of response and prepares the data for it. For example, returning a

PageResult doesn't generate any HTML at that point, it merely indicates that a view *should* be rendered. This is in keeping with the MVC design pattern in which it's the *view* that generates the response, not the *controller*.

> **TIP** The page handler is responsible for choosing what sort of response to send; the *view engine* in the MVC framework uses the result to generate the response.

It's also worth bearing in mind that page handlers should generally not be performing business logic directly. Instead, they should call appropriate services in the application model to handle requests. If a page handler receives a request to add a product to a user's cart, it shouldn't directly manipulate the database or recalculate cart totals, for example. Instead, it should make a call to another class to handle the details. This approach of separating concerns ensures your code stays testable and maintainable as it grows.

4.3.1 Accepting parameters to page handlers

Some requests made to page handlers will require additional values with details about the request. If the request is for a search page, the request might contain details of the search term and the page number they're looking at. If the request is posting a form to your application, for example a user logging in with their username and password, then those values must be contained in the request. In other cases, there will be no such values, such as when a user requests the homepage for your application.

The request may contain additional values from a variety of different sources. They could be part of the URL, the query string, headers, or in the body of the request itself. The middleware will extract values from each of these sources and convert them into .NET types.

> **DEFINITION** The process of extracting values from a request and converting them to .NET types is called *model binding*. I discuss model binding in chapter 6.

ASP.NET Core can bind two different targets in Razor Pages:

- *Method arguments*. If a page handler has method arguments, the values from the request are used to create the required parameters.
- *Properties marked with a* [BindProperty] *attribute*. Any properties marked with the attribute will be bound. By default, this attribute does nothing for GET requests.

Model bound values can be simple types, such as strings and integers, or they can be a complex type, as shown in the listing below. If any of the values provided in the request are *not* bound to a property or page handler argument, then the additional values will go unused.

Listing 4.3 Example Razor Page handlers

```
public class SearchModel : PageModel
{
    private SearchService _searchService;              #A
    public SearchModel(SearchService searchService)    #A
    {                                                    #A
        _searchService = searchService;                  #A
    }                                                    #A

    [BindProperty]                                       #B
    public BindingModel Input { get; set; }              #B
    public List<Product> Results { get; set; }           #C

    public void OnGet()                                  #D
    {                                                    #D
    }                                                    #D

    public IActionResult OnPost(int max)                  #E
    {
        if (ModelState.IsValid)                                      #F
        {                                                            #F
            Results = _searchService.Search (Input.SearchTerm, max);   #F
            return Page();                                           #F
        }                                                            #F
        return RedirectToPage("./Index");                              #G
    }                                                                #F
}
```

#A The SearchService is provided to the SearchModel for use in page handlers.
#B Properties decorated with the [BindProperty] attribute will be model bound.
#C Undecorated properties will not be model bound.
#D The page handler doesn't need to check if the model is valid. Returning void will render the view.
#E The max parameter in this page handler will be model bound using the values in the request.
#E If the request is valid, a view model is set and a PageResult is returned to render the view.
#F If the request was not valid, the method indicates the user should be redirected to the Index page.

In this example, the `OnGet` handler doesn't require any parameters, and the method is simple—it returns `void`, which means the associated Razor view will be rendered. It could also have returned a `PageResult`; the effect would have been the same. Note that this handler is for HTTP `GET` requests, so the `Input` property decorated with `[BindProperty]` is not bound.

> **TIP** **To bind properties for `GET` requests too, use the `SupportsGet` property of the attribute, for example:** `[BindProperty(SupportsGet = true)]`.

The `OnPost` handler, conversely, accepts a parameter `max` as an argument. In this case it's a simple type, `int`, but it could also be a complex object. Additionally, as this handler corresponds to an HTTP `POST` request, the `Input` property is also model bound to the request.

When an action method uses model bound properties or parameters, it should always check that the provided model is valid using `ModelState.IsValid`. The `ModelState` property is exposed as a property on the base `PageModel` class and can be used to check that all the bound properties and parameters are valid. You'll see how the process works in chapter 6 when you learn about validation.

Once a page handler establishes that the method parameters provided to an action are valid, it can execute the appropriate business logic and handle the request. In the case of the

OnPost handler, this involves calling the provided SearchService and setting the result on the Results property. Finally, the handler returns a PageResult by calling the base method

```
return Page();
```

If the model wasn't valid, then you don't have any results to display! In this example, the action returns a RedirectToPageResult using the RedirectToPage helper method. When executed, this result will send a 302 redirect response to the user, which will cause their browser to navigate to the Index Razor Page.

Note that the OnGet method returns void in the method signature, whereas the OnPost method returns an IActionResult. This is required in the OnPost method in order to allow the C# to compile (as the Page and RedirectToPage helper methods return different types), but it doesn't change the final behavior of the methods. You could just as easily have called Page in the OnGet method and returned an IActionResult and the behavior would be identical.

> **TIP** **If you're returning more than one type of result from a page handler, you'll need to ensure your method returns an** IActionResult.

In the next section we look in more depth at action results and what they're used for.

4.3.2 Returning responses with ActionResults

In the previous section, I emphasized that page handlers only decide *what type* or response to return, but they don't generate the response themselves. It's the IActionResult returned by a page handler which, when executed by the Razor Pages infrastructure using the view engine, will generate the response.

This approach is key to following the MVC design pattern. It separates the decision of what sort of response to send from the generation of the response. This allows you to easily test your action method logic to confirm the right sort of response is sent for a given input. You can then separately test that a given IActionResult generates the expected HTML, for example.

ASP.NET Core has many different types of IActionResult:

- PageResult—Generates an HTML view for an associated page in Razor Pages.
- ViewResult—Generates an HTML view for a given Razor view when using MVC controllers.
- RedirectToPageResult—Sends a 302 HTTP redirect response to automatically send a user to another page.
- RedirectResult—Sends a 302 HTTP redirect response to automatically send a user to a specified URL (doesn't have to be a Razor Page).
- FileResult—Returns a file as the response.
- ContentResult—Returns a provided string as the response.

- StatusCodeResult—Sends a raw HTTP status code as the response, optionally with associated response body content.
- NotFoundResult—Sends a raw 404 HTTP status code as the response.

Each of these, when executed by Razor Pages, will generate a response to send back through the middleware pipeline and out to the user.

> **TIP** When you're using Razor Pages, you generally won't use some of these action results, such as ContentResult and StatusCodeResult. It's good to be aware of them though, as you will likely use them if you are building Web APIs with MVC controllers.

In this section I give a brief description of the most common IActionResult classes that you'll use with Razor Pages.

PAGERESULT AND REDIRECTTOPAGERESULT

When you're building a traditional web application with Razor Pages, most of the time you'll be using the PageResult, which generates an HTML response using Razor. We'll look at how this happens in detail in chapter 7.

You'll also commonly use the various redirect-based results to send the user to a new web page. For example, when you place an order on an e-commerce website you typically navigate through multiple pages, as shown in figure 4.12. The web application sends HTTP redirects whenever it needs you to move to a different page, such as when a user submits a form. Your browser automatically follows the redirect requests, creating a seamless flow through the checkout process.

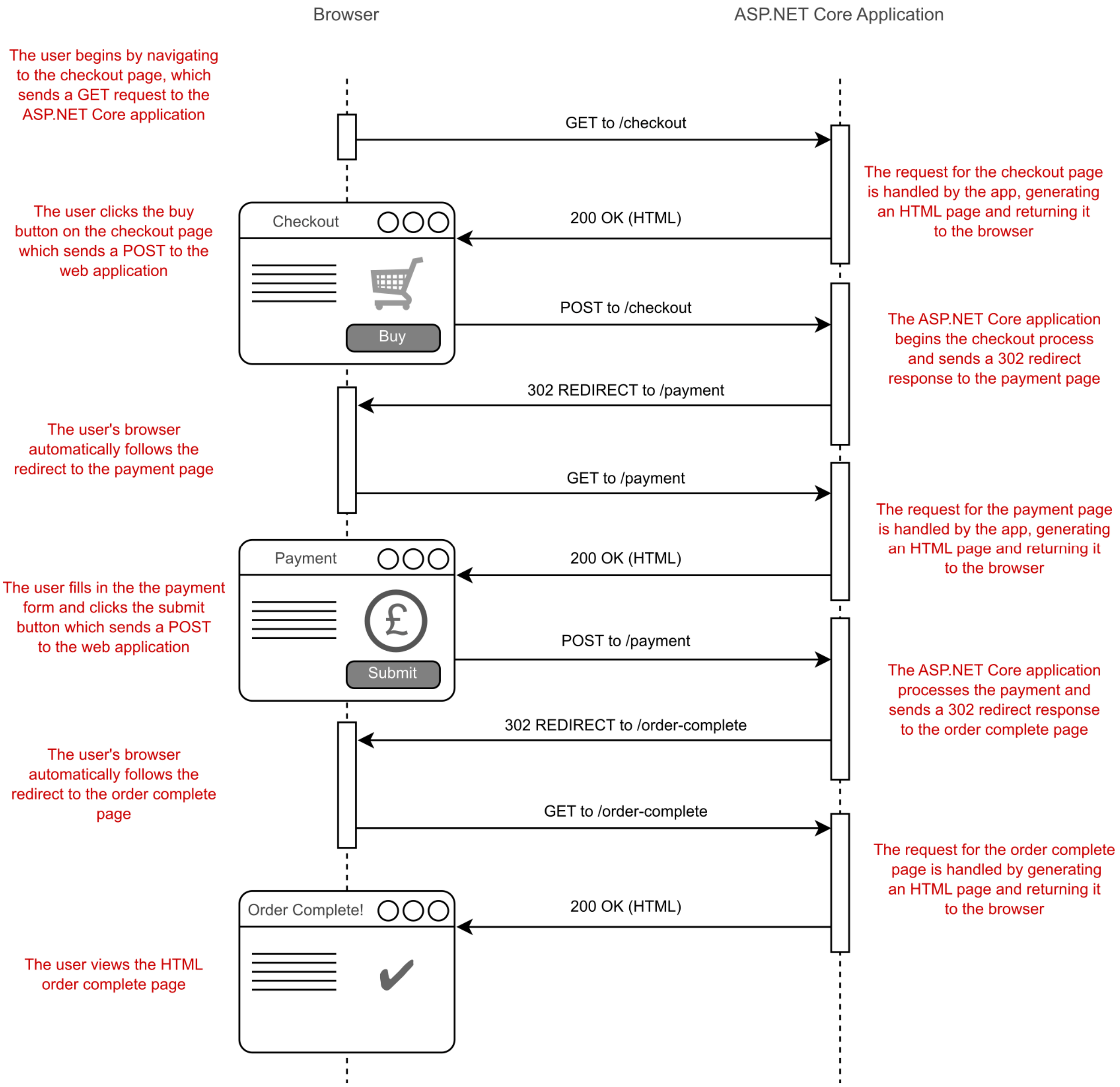

Browser
ASP.NET Core Application
The user begins by navigating to the checkout page, which sends a GET request to the ASP.NET Core application
GET to /checkout
The request for the checkout page is handled by the app, generating an HTML page and returning it to the browser
The user clicks the buy button on the checkout page which sends a POST to the web application
Checkout
Buy
200 OK (HTML)
POST to /checkout
The ASP.NET Core application begins the checkout process and sends a 302 redirect response to the payment page
302 REDIRECT to /payment
The user's browser automatically follows the redirect to the payment page
GET to /payment
The request for the payment page is handled by the app, generating an HTML page and returning it to the browser
Payment
£
Submit
200 OK (HTML)
The user fills in the the payment form and clicks the submit button which sends a POST to the web application
POST to /payment
The ASP.NET Core application processes the payment and sends a 302 redirect response to the order complete page
302 REDIRECT to /order-complete
The user's browser automatically follows the redirect to the order complete page
GET to /order-complete
The request for the order complete page is handled by generating an HTML page and returning it to the browser
Order Complete!
200 OK (HTML)
The user views the HTML order complete page

Figure 4.12 A typical POST, REDIRECT, GET flow through a website. A user sends their shopping basket to a checkout page, which validates its contents and redirects to a payment page without the user having to manually change the URL.

In this flow, whenever you return HTML you use a `PageResult`; when you redirect to a new page, you use a `RedirectToPageResult`.

NOTFOUNDRESULT AND STATUSCODERESULT

As well as HTML and redirect responses, you'll occasionally need to send specific HTTP status codes. If you request a page for viewing a product on an e-commerce application, and that product doesn't exist, a 404 HTTP status code is returned to the browser and you'll typically see a "Not found" web page. Razor Pages can achieve this behavior by returning a `NotFoundResult`, which will return a raw 404 HTTP status code. You could achieve a similar result using the `StatusCodeResult` and setting the status code returned explicitly to 404.

Note that the `NotFoundResult` doesn't generate any HTML; it only generates a raw 404 status code and returns it through the middleware pipeline. But, as discussed in the previous chapter, you can use the `StatusCodePagesMiddleware` to intercept this raw 404 status code after it's been generated and provide a user-friendly HTML response for it.

CREATING ACTIONRESULT CLASSES USING HELPER METHODS

`ActionResult` classes can be created and returned using the normal `new` syntax of C#:

```
return new PageResult()
```

However, the Razor Pages `PageModel` base class also provides a number of helper methods for generating responses. It's common to use the `Page` method to generate an appropriate `PageResult`, the `RedirectToPage` method to generate a `RedirectToPageResult`, or the `NotFound` method to generate a `NotFoundResult`.

> **TIP** **Most `ActionResult` classes have a helper method on the base `PageModel` class. They're typically named `Type`, and the result generated is called `TypeResult`. For example, the `StatusCode` method returns a `StatusCodeResult` instance.**

As discussed earlier, the act of *returning* an `IActionResult` doesn't immediately generate the response—it's the *execution* of an `IActionResult` by the Razor Pages infrastructure, which occurs outside the action method. After producing the response, Razor Pages returns it to the middleware pipeline. From there, it passes through all the registered middleware in the pipeline, before the ASP.NET Core web server finally sends it to the user.

By now, you should have an overall understanding of the MVC design pattern and how it relates to ASP.NET Core and Razor Pages. The page handler methods on a Razor Page are invoked in response to given requests and are used to select the type of response to generate by returning an `IActionResult`.

It's important to remember that the MVC and Razor Pages infrastructure in ASP.NET Core runs as part of the `EndpointMiddleware` pipeline, as you saw in the previous chapter. Any response generated, whether a `PageResult` or a `RedirectToPageResult`, will pass back through the middleware pipeline, giving a potential opportunity for middleware to observe the response before the web server sends it to the user.

An aspect I've only vaguely touched on is how the `RoutingMiddleware` decides which Razor Page and handler to invoke for a given request. If you had to have a Razor Page for *every* URL in an app, it would be difficult to have, for example, a different page per product in an e-shop—every product would need its own Razor Page! Handling this and other scenarios is the role of the routing infrastructure and is a key part of ASP.NET Core. In the next chapter, you'll see how to define routes, how to add constraints to your routes, and how they deconstruct URLs to match a single Razor Page handler.

4.4 Summary

- The MVC design pattern allows for a separation of concerns between the business logic of your application, the data that's passed around, and the display of data in a response.
- Razor Pages are built on the ASP.NET Core MVC framework and use many of the same primitives. They use conventions and a different project layout to optimize for page-based scenarios.
- MVC *controllers* contain multiple *action methods,* typically grouped around a high-level entity. Razor Pages groups all the *page handlers* for a single page in one place, grouping around a page/feature instead of an entity.
- Each Razor Page is equivalent to a mini controller focused on a single page, and each Razor Page handler corresponds to a separate action method.
- Razor Pages should inherit from the `PageModel` base class.
- A single Razor Page handler is selected based on the incoming request's URL, the HTTP verb, and the request's query string, in a process called *routing*.
- Page handlers should generally delegate to services to handle the business logic required by a request, instead of performing the changes themselves. This ensures a clean separation of concerns that aids testing and improves application structure.
- Page handlers can have parameters whose values are taken from properties of the incoming request in a process called *model binding*. Properties decorated with `[BindProperty]` can also be bound to the request.
- By default, properties decorated with `[BindProperty]` are not bound for GET requests. To enable binding, use `[BindProperty(SupportsGet = true)]`.
- Page Handlers can return a `PageResult` or `void` to generate an HTML response.
- You can send users to a new Razor Page using a `RedirectToPageResult`.
- The `PageModel` base class exposes many helper methods for creating an `ActionResult`.

5

Mapping URLs to Razor Pages using routing

This chapter covers

- **Mapping URLs to Razor Pages**
- **Using constraints and default values to match URLs**
- **Generating URLs from route parameters**

In chapter 4, you learned about the MVC design pattern, and how ASP.NET Core uses it to generate the UI for an application using Razor Pages. Razor Pages contain page handlers which act as “mini controllers” for a request. The page handler calls the application model to retrieve or save data. The handler then passes data from the application model to the Razor view, which generates an HTML response.

Although not part of the MVC design pattern per-se, one crucial part of Razor Pages is selecting which Razor Page to invoke in response for a given request. This process is called *routing* and is the focus of this chapter.

This chapter begins by identifying the need for routing and why it’s useful. You’ll learn about the endpoint routing system introduced in ASP.NET Core 3.0, see several examples of routing techniques, and explore the separation routing can bring between the layout of your Razor Page files and the URLs you expose.

The bulk of this chapter focuses on how to use routing with Razor Pages to create dynamic URLs, so that a single Razor Page can handle requests to multiple URLs. I’ll show how to build powerful route templates and give you a taste of the available options.

In section 5.5, I describe how to use the routing system to *generate* URLs, which you can use to create links and redirect requests for your application. One of the benefits of using a routing system is that it decouples your Razor Pages from the underlying URLs that are used to execute them. You can use URL generation to avoid littering your code with hardcoded URLs like `/Product/View/3`. Instead, you can generate the URLs at runtime, based on the routing system. The benefit of this is that it makes changing the URL configuration for a Razor Page easier. Instead of having to hunt down everywhere you used the Razor Page's URL, the URLs will be automatically updated for you, with no other changes required.

I finish the chapter by describing how you can customize the conventions Razor Pages uses, giving you complete control over the URLs your application uses. You'll see how to change the built-in conventions, such as using lowercase for your URLs, as well as how to write your own convention and apply it globally to your application.

By the end of this chapter, you should have a much clearer understanding of how an ASP.NET Core application works. You can think of routing as the glue that ties the middleware pipeline to Razor Pages and the MVC framework. With middleware, Razor Pages, and routing under your belt, you'll be writing web apps in no time!

5.1 What is routing?

In this section, I teach about routing. Routing is the process of mapping an incoming request to a method that will handle it. You can use routing to control the URLs you expose in your application. You can also use routing to enable powerful features like mapping multiple URLs to the same Razor Page, and automatically extracting data from a request's URL.

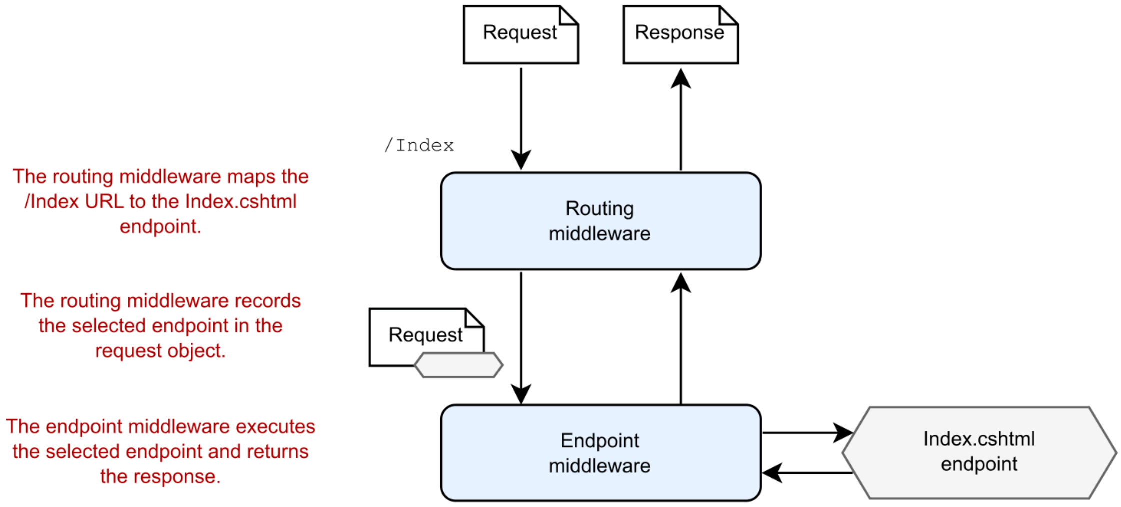
In chapter 3, you saw that an ASP.NET Core application contains a middleware pipeline, which defines the behavior of your application. Middleware is well suited to handling both cross-cutting concerns, such as logging and error handling, and narrowly focused requests, such as requests for images and CSS files.

To handle more complex application logic, you'll typically use the `EndpointMiddleware` at the end of your middleware pipeline as you saw in chapter 4. This middleware can handle an appropriate request by invoking a method, known as a page handler on a Razor Page or an action method on an MVC controller, and using the result to generate a response.

One aspect that I glossed over in chapter 4 was *how* to select which Razor Page or action method to execute when you receive a request. What makes a request "appropriate" for a given Razor Page handler? The process of mapping a request to a handler is called *routing*.

> **DEFINITION** ***Routing*** **in ASP.NET Core is the process of mapping an incoming HTTP request to a specific handler. In Razor Pages, the handler is a page handler method in a Razor Page. In MVC, the handler is an action method.**

At this point you've already seen several simple applications built with Razor Pages in previous chapters, and so you've already seen routing in action, even if you didn't realize it at the time. Even a simple URL path—for example, `/Index`—uses routing to determine that Index.cshtml Razor Page should be executed, as shown in figure 5.1.



The routing middleware maps the /Index URL to the Index.cshtml endpoint.

The routing middleware records the selected endpoint in the request object.

The endpoint middleware executes the selected endpoint and returns the response.

Figure 5.1 The router compares the request URL against a list of configured route templates to determine which action method to execute.

On the face of it, that seems pretty simple. You may be wondering why I need a whole chapter to explain that obvious mapping. The simplicity of the mapping in this case belies how powerful routing can be. If this "file layout-based" approach were the only one available, you'd be severely limited in the applications you could feasibly build.

For example, consider an eCommerce application that sells multiple products. Each product needs to have its own URL, so if you were using a purely file layout-based routing system, you'd only have two options:

- *Use a different Razor Page for every product in your product range.* That would be completely unfeasible for almost *any* realistically sized product range.
- Use a single Razor Page and use the query string to differentiate between products. This is much more practical, but would end up with somewhat ugly URLs, like `"/product?name=big-widget"`, or `"/product?id=12"`.

DEFINITION **The *query string* is part of a URL that contains additional data that doesn't fit in the path. It isn't used by the routing infrastructure for identifying which action to execute, but can be used for model binding, as you'll see in chapter 6.**

With routing, you can have a *single* Razor Page that can handle *multiple* URLs, without having to resort to ugly query strings. From the point of the view of the Razor Page, the query string and routing approaches are very similar—the Razor Page dynamically displays the results for the correct product as appropriate. The difference is that with routing, we can completely customize the URLs, as shown in figure 5.2. This gives much more flexibility and can be important in real life applications for search engine optimization (SEO) reasons[16].

[16] Importantly, you can properly encode the hierarchy of your site in your URLs, as described in Google's SEO starter guide: https://support.google.com/webmasters/answer/7451184

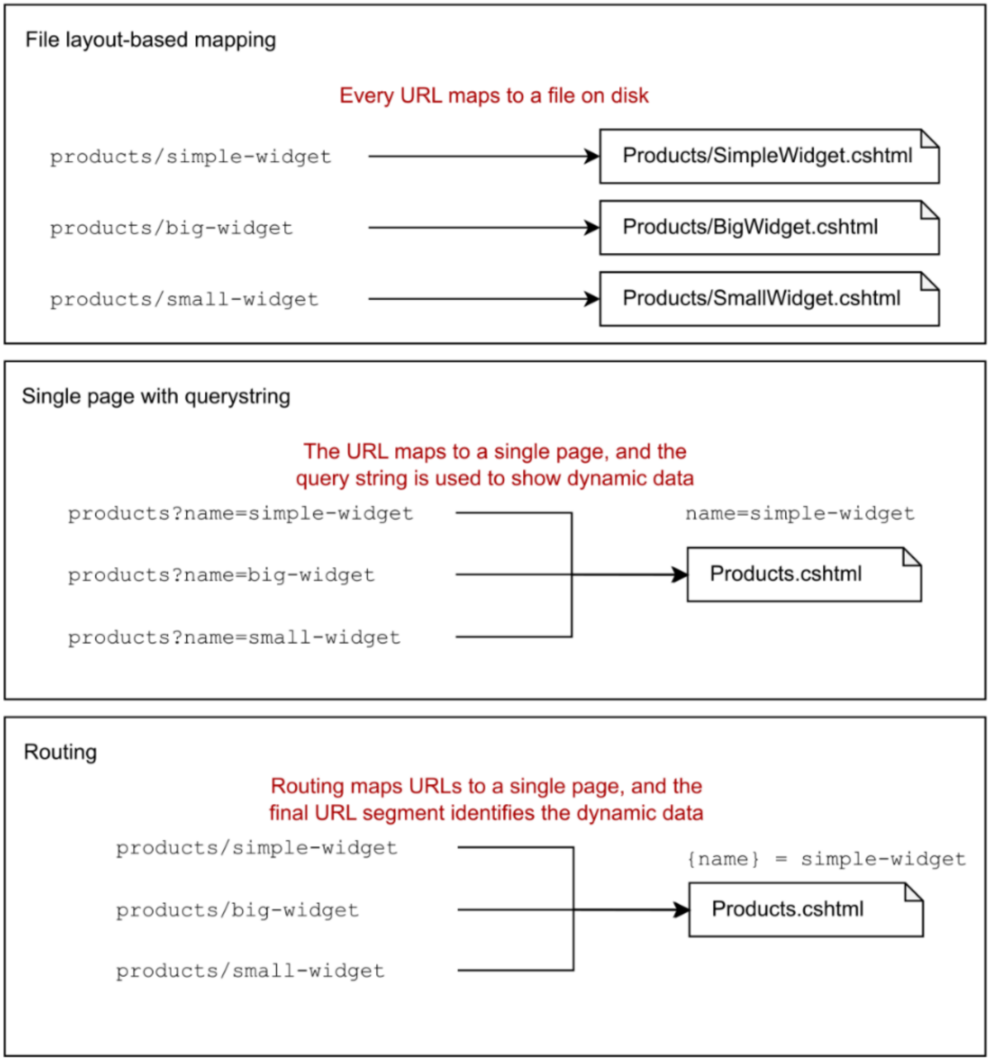

File layout-based mapping
Every URL maps to a file on disk
products/simple-widget
Products/SimpleWidget.cshtml
products/big-widget
Products/BigWidget.cshtml
products/small-widget
Products/SmallWidget.cshtml
Single page with querystring
The URL maps to a single page, and the query string is used to show dynamic data
products?name=simple-widget
products?name=big-widget
products?name=small-widget
name=simple-widget
Products.cshtml
Routing
Routing maps URLs to a single page, and the final URL segment identifies the dynamic data
products/simple-widget
products/big-widget
products/small-widget
{name} = simple-widget
Products.cshtml

Figure 5.2 If you use file-layout based mapping, you need a different Razor Page for every product in your product range. With routing, multiple URLs map to a single Razor Page, and a dynamic parameter captures the difference in the URL.

As well as enabling dynamic URLs, routing fundamentally decouples the URLs in your application from the file names of your Razor Pages. For example, say you had a currency converter application, with a Razor Page in your project located at the path Pages/Rates/View.cshtml, which is used to view the exchange rate for a currency, say USD. By default, this might correspond to the `/rates/view/1` URL for users. This would work fine, but it doesn't tell users much—which currency will this show? Will it be a historical view or the current rate?

Luckily, with routing it's easy to modify your exposed URLs without having to change your Razor Page file names or locations. Depending on your routing configuration, you could set the URL pointing to the View.chstml Razor Page to any of the following:

- `/rates/view/1`
- `/rates/view/USD`
- `/rates/current-exchange-rate/USD`
- `/current-exchange-rate-for-USD`

I know which of these I'd most like to see in the URL bar of my browser, and which I'd be most likely to click! This level of customization isn't often necessary, and the default URLs are normally the best option in the long run, but it's very useful to have the capability to customize the URLs when you need it.

In the next section we'll look at how routing works in practice in ASP.NET Core.

5.2 Routing in ASP.NET Core

In this section we look at:

- The endpoint routing system introduced in ASP.NET Core 3.0.
- The two supported routing approaches: convention-based routing and attribute routing.
- How routing works with Razor Pages.

Routing has been a part of ASP.NET Core since its inception, but in ASP.NET Core 3.0 it went through some big changes. In ASP.NET Core 2.0 and 2.1, routing was restricted to Razor Pages and the ASP.NET Core MVC framework. There was no dedicated routing middleware in your middleware pipeline—routing happened only within Razor Pages or MVC components.

Given that most of the logic of your application is implemented in Razor Pages, only using routing for Razor Pages was fine for the most part. Unfortunately, restricting routing to the MVC infrastructure made some things a bit messy. It meant some cross-cutting concerns, like authorization, were restricted to the MVC infrastructure and were hard to use from other middleware in your application. That restriction caused inevitable duplication, which wasn't ideal.

In ASP.NET Core 3.0, a new routing system was introduced, *endpoint routing*. Endpoint routing makes the routing system a more fundamental feature of ASP.NET Core, and no longer ties it to the MVC infrastructure. Razor Pages and MVC still rely on endpoint routing, but now other middleware can use it too.

In this section I cover

- How endpoint routing works in ASP.NET Core
- The two types of routing available: convention-based routing and attribute routing
- How routing works for Razor Pages

At the end of this section you should have a good overview of how routing in ASP.NET Core works with Razor Pages.

5.2.1 Using endpoint routing in ASP.NET Core

Endpoint routing is fundamental to all but the simplest ASP.NET Core apps. It's implemented using two pieces of middleware, which you've seen previously:

- `EndpointMiddleware`. You use this middleware to *register* the "endpoints" in routing the system when you start your application. The middleware *executes* one of the endpoints at runtime.
- `EndpointRoutingMiddleware`. This middleware chooses *which* of the endpoints registered by the `EndpointMiddleware` should execute for a given request at runtime. To make it easier to distinguish between the two types of middleware, I'll be referring to this middleware as the `RoutingMiddleware` throughout this book.

The `EndpointMiddleware` is where you configure all the *endpoints* in your system. This is where you register your Razor Pages and MVC controllers, but you can also register additional handlers that fall outside of the MVC framework, such as health-check endpoints that confirm your application is still running.

> **DEFINITION** An *endpoint* in ASP.NET Core is some handler that returns a response. Each endpoint is associated with a URL pattern. Razor Page handlers and MVC controller action methods typically make up the bulk of the endpoints in an application, but you can also use simple middleware as an endpoint, or a health-check endpoint.

To register endpoints in your application, call `UseEndpoints` in the `Configure` method of Startup.cs. This method takes a configuration lambda action that defines the endpoints in your application, as shown in the following listing. You can automatically register all the Razor Pages in your application using extensions such as `MapRazorPages`. Additionally, you can register other endpoints explicitly using methods such as `MapGet`.

Listing 5.1 Registering multiple endpoints in Startup.Configure

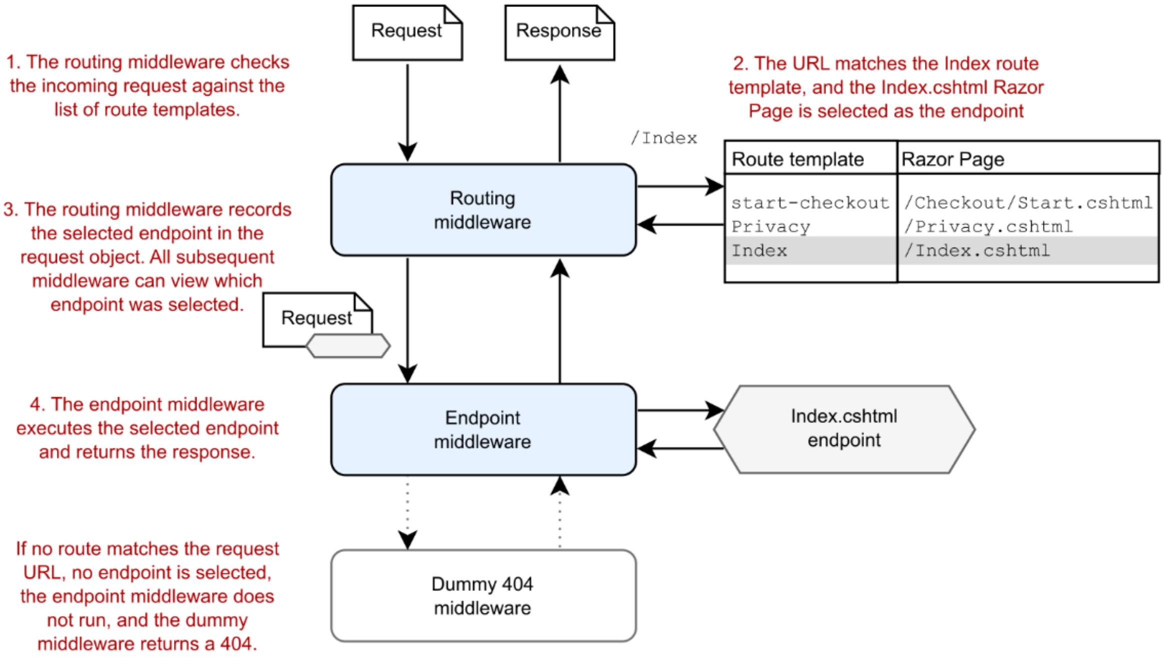
```
public void Configure(IApplicationBuilder app)
{
    app.UseRouting();                                          #A

    app.UseEndpoints(endpoints =>                              #B
    {
        endpoints.MapRazorPages();                             #C
        endpoints.MapHealthChecks("/healthz");                 #D
        endpoints.MapGet("/test", async context =>             #E
        {                                                      #E
            await context.Response.WriteAsync("Hello World!"); #E
        });                                                    #E
    });
}
```

#A Add the EndpointRoutingMiddleware to the middleware pipeline
#B Add the EndpointMiddleware to the pipeline and provide a configuration lambda
#C Register all the Razor Pages in your application as an endpoint
#D Register a health check endpoint at the route /healthz
#E Register an endpoint inline that returns "Hello World!" at the route /test

Each endpoint is associated with a *route template* that defines which URLs the endpoint should match. You can see two route templates, `"/healthz" and "/test"`, in the previous listing.

> **DEFINITION** **A *route template* is a URL pattern that is used to match against request URLs. They're strings of fixed values, like `"/test"` in the previous listing. They can also contain placeholders for variables, as you'll see in section 5.3.**

The `EndpointMiddleware` stores the registered routes and endpoints in a dictionary, which it shares with the `RoutingMiddleware`. At runtime, the `RoutingMiddleware` compares an incoming request to the routes registered in the dictionary. If the `RoutingMiddleware` finds a matching endpoint, then it makes a note of which endpoint was selected and attaches that to the request's `HttpContext` object. It then calls the next middleware in the pipeline. When the request reaches the `EndpointMiddleware`, the middleware checks to see which endpoint was selected, and executes it, as shown in figure 5.3.



1. The routing middleware checks the incoming request against the list of route templates.

2. The URL matches the Index route template, and the Index.cshtml Razor Page is selected as the endpoint

Route template	Razor Page
start-checkout	/Checkout/Start.cshtml
Privacy	/Privacy.cshtml
Index	/Index.cshtml

3. The routing middleware records the selected endpoint in the request object. All subsequent middleware can view which endpoint was selected.

4. The endpoint middleware executes the selected endpoint and returns the response.

If no route matches the request URL, no endpoint is selected, the endpoint middleware does not run, and the dummy middleware returns a 404.

Figure 5.3 Endpoint routing uses a two-step process. The `RoutingMiddleware` selects which endpoint to execute, and the `EndpointMiddleware` executes it. If the request URL doesn't match a route template, the endpoint middleware will not generate a response.

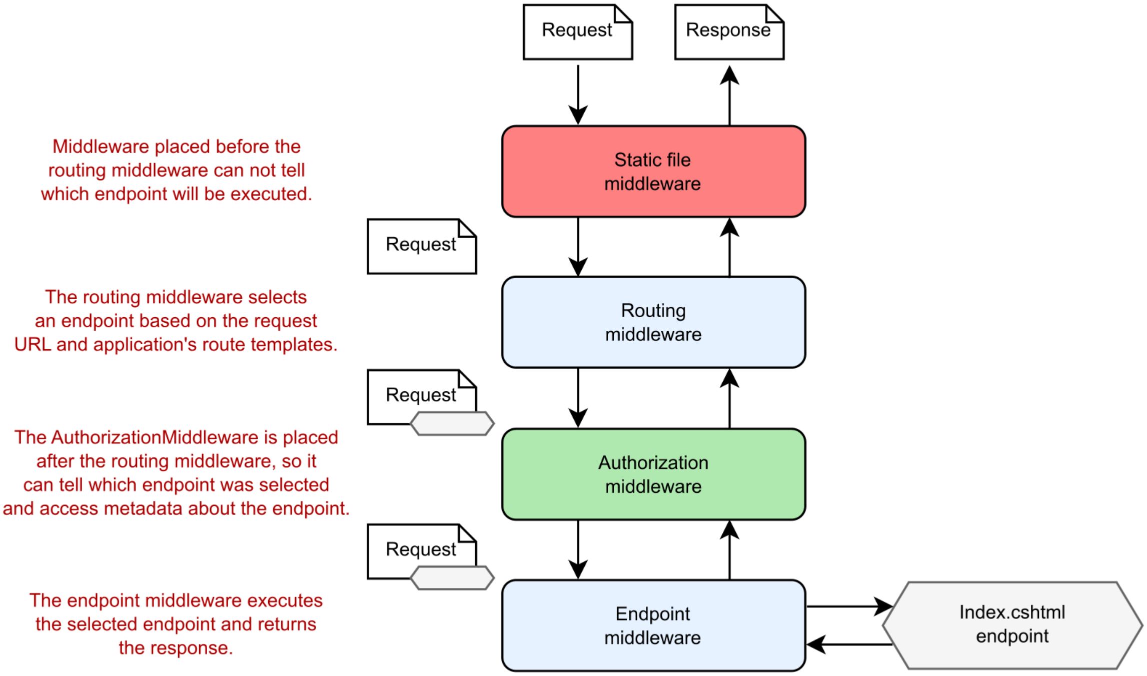
If the request URL *doesn't* match a route template, then the `RoutingMiddleware` doesn't select an endpoint, but the request still continues down the middleware pipeline. As no endpoint is selected, the `EndpointMiddleware` silently ignores the request, and passes it to the next middleware in the pipeline. The `EndpointMiddleware` is typically the final middleware in the pipeline, so the "next" middleware is normally the "dummy" middleware that always returns a `404 Not Found` response, as you saw in chapter 3.

> **TIP** **If the request URL does not match a route template, no endpoint is selected or executed. The whole middleware pipeline is still executed, but typically a 404 response is returned when the request reaches the "dummy" 404 middleware.**

The advantage of having two separate pieces of middleware to handle this process might not be obvious at first blush. Figure 5.3 hinted at the main benefit—all middleware placed after the `RoutingMiddleware` can see which endpoint is *going* to be executed before it is.

> **NOTE** **Only middleware placed *after* the `RoutingMiddleware` can detect which endpoint is going to be executed.**

Figure 5.4 shows a more realistic middleware pipeline, where middleware is placed both *before* the `RoutingMiddleware` and *between* the `RoutingMiddleware` and the `EndpointMiddleware`.



Middleware placed before the routing middleware can not tell which endpoint will be executed.

The routing middleware selects an endpoint based on the request URL and application's route templates.

The AuthorizationMiddleware is placed after the routing middleware, so it can tell which endpoint was selected and access metadata about the endpoint.

The endpoint middleware executes the selected endpoint and returns the response.

Figure 5.4 Middleware placed before the routing middleware doesn't know which endpoint the routing middleware will select. Middleware placed between the routing middleware and the endpoint middleware can see the selected endpoint.

The `StaticFileMiddleware` in figure 5.4 is placed *before* the `RoutingMiddleware`, so it executes before an endpoint is selected. Conversely, the `AuthorizationMiddleware` is placed *after* the `RoutingMiddleware`, so it can tell that the Index.cshtml Razor Page endpoint will be executed eventually. In addition, it can access certain metadata about the endpoint, such as its name and what the required permissions are to access the Razor Page.

> **TIP** The `AuthorizationMiddleware` needs to know which endpoint will be executed, so it must be placed ***after*** the `RoutingMiddleware` and ***before*** the `EndpointMiddleware` in your middleware pipeline. I discuss authorization in more detail in chapter 15.

It's important to remember the different roles of the two types of routing middleware when building your application. If you have a piece of middleware that needs to know which endpoint (if any) a given request will execute, then you need to make sure you place it after the `RoutingMiddleware` and before the `EndpointMiddleware`.

We've covered how the `RoutingMiddleware` and `EndpointMiddleware` interact to provide routing capabilities in ASP.NET Core, but we haven't yet looked at *how* the `RoutingMiddleware` matches the request URL to an endpoint. In the next section we look at the two different approaches used in ASP.NET Core.

5.2.2 Convention-based routing vs attribute routing

Routing is a key part of ASP.NET Core, as it maps the incoming request's URL to a specific endpoint to execute. You have two different ways to define these URL-endpoint mappings in your application:

- Using global, convention-based routing.
- Using attribute routing.

Which approach you use will typically depend on whether you're using Razor Pages or MVC controllers, and whether you're building an API or a website (using HTML). These days, I lean heavily towards attribute routing, as you'll see shortly.

Convention-based routing is defined globally for your application. You can use convention-based routes to map endpoints (MVC controller actions) in your application to URLs, but your MVC controllers must adhere strictly to the conventions you define. Traditionally, applications using MVC controllers to generate HTML tend to use this approach to routing. The downside to this approach is it makes customizing the URLs for a subset of controllers and actions more difficult.

Alternatively, you can use attribute-based routes to tie a given URL to a specific endpoint. For MVC controllers, this involves placing `[Route]` attributes on the action methods themselves, hence the term *attribute-routing*. This provides a lot more flexibility, as you can explicitly define what the URL for each action method should be. This approach is generally more verbose than the convention-based approach, as it requires applying attributes to *every* action method in your application. Despite this, the additional flexibility it provides can be very useful, especially when building Web APIs.

Somewhat confusingly, Razor Pages uses *conventions* to generate *attribute routes*! In many ways this combination gives the best of both worlds—you get the predictability and terseness of convention-based routing with the easy customization of attribute routing. There are trade-offs to each of the approaches, as shown in table 5.1.

Table 5.1 The advantages and disadvantages of the routing styles available in ASP.NET Core

Routing style	Typical use	Advantages	Disadvantages
Convention-based routes	HTML generating MVC controllers	Very terse definition in one location in your application. Forces a consistent layout of MVC	Routes are defined in different place to your controllers. Overriding the route conventions

		controllers.	can be tricky and error prone. Adds an extra layer of indirection when routing a request.
Attribute routes	Web API MVC controllers	Gives complete control over route templates for every action. Routes are defined next to the action they execute.	Verbose compared to convention-based routing Can be easy to over-customize route templates Route templates are scattered throughout your application, rather than in one location.
Convention-based generation of attribute routes	Razor Pages	Encourages consistent set of exposed URLs. Terse when you stick to the conventions Easily override the route template for a single page Customize conventions globally to change exposed URLs	Possible to over-customize route templates You must calculate what the route template for a page is, rather than it being explicitly defined in your app

So which approach should you use? My opinion is that convention-based routing is not worth the effort in 99% of cases, and that you should stick to attribute routing. If you're following my advice of using Razor Pages, then you're already using attribute routing under the covers. Also, if you're creating APIs using MVC controllers, then attribute routing is the best option, and is the recommended approach.

The only scenario where convention-based routing is used traditionally is if you're using MVC controllers to generate HTML. But if you are following my advice from chapter 4, you'll be using Razor Pages for HTML generating applications, and only falling back to MVC controllers when completely necessary. For consistency, in that scenario I would still stick with attribute routing.

> **NOTE** For the reasons above, this book focuses on attribute routing. Virtually of the features described in this section also apply to convention-based routing. For details on convention-based routing see the documentation: https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing.

Whichever technique you use, you'll define your application's expected URLs using *route templates*. These define the pattern of the URL you're expecting, with placeholders for the parts that may vary.

> **DEFINITION** *Route templates* define the structure of known URLs in your application. They're strings with placeholders for variables that can contain optional values.

A single route template can match many different URLs. For example, the /customer/1 and /customer/2 URLs would both be matched by the "customer/{id}" route template. The route template syntax is powerful and contains many different features that are controlled by splitting a URL into multiple *segments*.

> **DEFINITION** A *segment* is a small contiguous section of a URL. It's separated from other URL segments by at least one character, often by the / character. Routing involves matching the segments of a URL to a route template.

For each route template, you can define

- Specific, expected strings
- Variable segments of the URL
- Optional segments of a URL
- Default values when an optional segment isn't provided
- Constraints on segments of a URL, for example, ensuring that it's numeric

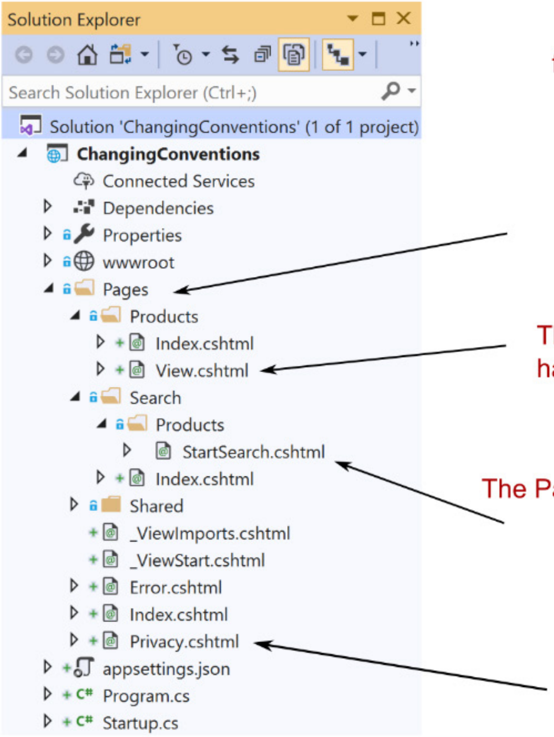
Most applications will use a variety of these features, but you often only use one or two features here and there. For the most part, the default convention-based attribute route templates generated by Razor Pages will be all you need. In the next section we look at those conventions, and how routing maps a request's URL to a Razor Page in detail.

5.2.3 Routing to Razor Pages

As I mentioned in the section 5.2.2, Razor Pages uses attribute routing by creating route templates based on conventions. ASP.NET Core creates a route template for every Razor Page in your app during app startup, when you call MapRazorPages in the Configure method of Startup.cs:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
});
```

For every Razor Page in your application, the framework uses the path of the Razor Page file relative to the Razor Pages root directory (Pages/), excluding the file extension (cshtml). For example, if you have a Razor Page located at the path Pages/Products/View.cshtm, the framework creates a route template with the value "Products/View", as shown in figure 5.5.



Route templates are based on the file path relative to the Razor Pages root directory.

The Razor Pages root directory is called Pages.

The Pages/Products/View.cshtml page has a route template of Products/View.

The Pages/Search/Products/StartSearch.cshtml page has a route template of Search/Products/StartSearch.

The Pages/Privacy.cshtml page has a route template of Privacy.

Figure 5.5 By default, route templates are generated for Razor Pages based on the path of the file relative to the root directory, Pages.

Requests to the URL `/products/view` match the route template `"Products/View"`, which in turn corresponds to the View.cshtml Razor Page. The `RoutingMiddleware` selects the View.cshtml Razor Page as the endpoint for the request, and the `EndpointMiddleware` executes the page's handler once the request reaches it in the middleware pipeline.

> **TIP** Routing is not case sensitive, so the request URL does not need to have the same URL casing as the route template to match.

In chapter 4, you learned that Razor Page handlers are the methods that are invoked on a Razor Page. When we say, "a Razor Page is executed" we really mean "an instance of the Razor Page's `PageModel` is created, and a page handler on the model is invoked". Razor Pages can have multiple page handlers, so once the `RoutingMiddleware` selects a Razor Page, the `EndpointMiddleware` still needs to choose which handler to execute. You'll learn how the framework selects which page handler to invoke in section 5.6.

By default, each Razor Page creates a single route template based on its file path. The exception to this rule is for Razor Pages that are called Index.cshtml. Index.cshtml pages create *two* route templates, one ending with `"Index"`, the other without an ending. For example, if you have a Razor Page at the path Pages/ToDo/Index.cshtml, that will generate two route templates:

- `"ToDo"`
- `"ToDo/Index"`

When either of these routes are matched, the same Index.cshtml Razor Page is selected. For example, if your application is running at the URL https://example.org, you can view the page by executing https://example.org/ToDo or https://example.org/ToDo/Index.

As a final example, consider the Razor Pages created by default when you create a Razor Pages application using Visual Studio or by running `dotnet new web` using the .NET CLI, as we did in chapter 2. The standard template includes three Razor Pages in the Pages directory:

- Pages/Error.cshtml
- Pages/Index.cshtml
- Pages/Privacy.cshtml

That creates a collection of four routes for the application, defined by the following templates:

- `""` maps to Index.cshtml
- `"Index"` maps to Index.cshtml
- `"Error"` maps to Error.cshtml
- `"Privacy"` maps to Privacy.cshtml

At this point, routing probably feels laughably trivial, but this is just the basics that you get "for free" with the default Razor Pages conventions, which is often sufficient for a large portion of any website. At some point though, you'll find you need something more dynamic, such as an ecommerce site where you want each product to have its own URL, but which maps to a single Razor Page. This is where route templates and route data come in and show the real power of routing.

5.3 Customizing Razor Page route templates

The route templates for a Razor Page are based on the file path by default, but you're also able to customize the final template for each page, or even to replace it entirely. In this section I show how to customize the route templates for individual pages, so you can customize your application's URLs, and map multiple URLs to a single Razor Page.

Route templates have a rich, flexible syntax, but a simple example is shown in figure 5.6.

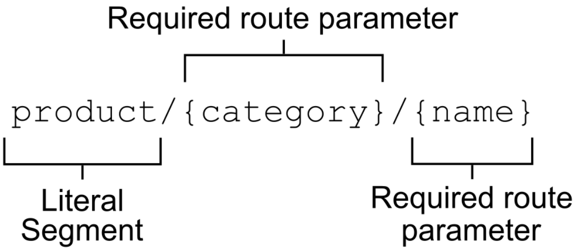


Figure 5.6 A simple route template showing a literal segment and two required route parameters

The routing middleware parses a route template by splitting it into a number of *segments*. A segment is typically separated by the `/` character, but it can be any valid character. Each segment is either

- *A literal value*—For example, `product` in figure 5.6
- *A route parameter*—For example, `{category}` and `{name}` in figure 5.6

Literal values must be matched exactly (ignoring case) by the request URL. If you need to match a particular URL exactly, you can use a template consisting only of literals. This is the default case for Razor Pages, as you saw in section 5.2.3; each Razor Page consists of a series of literal segments, for example `"ToDo/Index"`.

Imagine you have a contact page in your application at the path Pages/About/Contact.cshtml. The route template for this page is `"About/Contact"`. This route template consists of only literal values, and so only matches the exact URL. None of the following URLs would match this route template:

- `/about`
- `/about-us/contact`
- `/about/contact/email`
- `/about/contact-us`

Route parameters are sections of a URL that may vary, but will still be a match for the template. They are defined by giving them a name, and placing them in braces, such as `{category}` or `{name}`. When used in this way, the parameters are required, so there must be a segment in the request URL that they correspond to, but the value can vary.

> **TIP** **Some words can't be used as names for route parameters:** `area`, `action`, `controller`, `handler`, **and** `page`.

The ability to use route parameters gives you great flexibility. For example, the simple route template `"{category}/{name}"` could be used to match all the product-page URLs in an ecommerce application, such as:

- /bags/rucksack-a—Where category=bags and name=rucksack-a
- /shoes/black-size9—Where category=shoes and name=black-size-9

But note that this template would *not* map the following URLs:

- /socks/—No name parameter specified
- /trousers/mens/formal—Extra URL segment, formal, not found in route template

When a route template defines a route parameter, and the route matches a URL, the value associated with the parameter is captured and stored in a dictionary of values associated with the request. These *route values* typically drive other behavior in the Razor Page, such as model binding.

> **DEFINITION** ***Route values*** **are the values extracted from a URL based on a given route template. Each route parameter in a template will have an associated route value and is stored as a string pair in a dictionary. They can be used during model binding, as you'll see in chapter 6.**

Literal segments and route parameters are the two cornerstones of ASP.NET Core route templates. With these two concepts, it's possible to build all manner of URLs for your application. But how can you customize a Razor Page to use one of these patterns?

5.3.1 Adding a segment to a Razor Page route template

To customize the Razor Page route template, you update the @page directive at the top of the Razor Page's cshtml file. This directive must be the first thing in the Razor Page file for the page to be registered correctly.

> **NOTE You must include the @page directive at the top of a Razor Page's cshtml file. Without it, ASP.NET Core will not treat the file as a Razor Page, and you will not be able to view the page.**

To add an additional segment to a Razor Page's route template, add a space followed by the desired route template, after the @page statement. For example, to add "Extra" to a Razor Page's route template, use

```
@page "Extra"
```

This *appends* the provided route template to the default template generated for the Razor Page. For example, the default route template for the Razor Page at Pages/Privacy.html is "Privacy". With the directive above, the new route template for the page would be "Privacy/Extra".

> **NOTE The route template provided in the @page directive is *appended* to the end of the default template for the Razor Page.**

The most common reason for customizing a Razor Page's route template like this is to add a *route parameter*. For example, you could have a single Razor Page for displaying the products in an ecommerce site at the path Pages/Products.cshtml, and use a route parameter in the `@page` directive

```
@page "{category}/{name}"
```

This would give a final route template of `Products/{category}/{name}` which would match all of the following URLs:

- `/products/bags/white-rucksack`
- `/products/shoes/black-size9`
- `/Products/phones/iPhoneX`

It's very common to *add* route segments to the Razor Page template like this, but what if that's not enough? Maybe you don't want to have the `/products` segment at the start of the above URLs, or you want to use a completely custom URL for a page. Luckily that's just as easy to achieve.

5.3.2 Replacing a Razor Page route template completely

You'll be most productive working with Razor Pages if you can stick to the default routing conventions where possible, adding additional segments for route parameters where necessary. But sometimes you just need more control. That's often the case for "important" pages in your application, such as the checkout page for an ecommerce application, or even the product pages, as you saw in the previous section.

To specify a custom route for a Razor Page, prefix the route with `/` in the `@page` directive. For example, to remove the `"product/"` prefix from the route templates in the previous section, use the directive:

```
@page "/{category}/{name}"
```

Note that this directive includes the `"/"` at the start of the route, indicating that this is a *custom* route template, instead of an *addition*. The route template for this page will be `"{category}/{name}"`, no matter which Razor Page it is applied to.

Similarly, you can create a "static" custom template for a page by starting the template with a `"/"` and using only literal segments. For example:

```
@page "/checkout"
```

Wherever your place your checkout Razor Page within the Pages folder, using this directive ensures it always has the route template `"checkout"`, so it will always match the request URL `/checkout`.

It's important to note that when you customize the route template for a Razor Page, both when appending to the default and when replacing it with a custom route, the *default template is no longer valid*. For example, if you use the `"checkout"` route template above on a Razor

Page located at Pages/Payment.cshtml, you can *only* access it using the URL `/checkout`; the URL `/Payment` is no longer valid and will not execute the Razor Page.

> **TIP** **Customizing the route template for a Razor Page using the `@page` directive *replaces* the default route template for the page. In section 5.7 I show how you can add additional routes while preserving the default route template.**

In this section you learned how to customize the route template for a Razor Page. In the next section we look in more depth at the route template syntax and some of the other features available.

5.4 Exploring the route template syntax

As well as the basic elements of literals and route parameter segments, route templates have extra features that give you more control over your application's URLs. These features let you have optional URL segments, provide default values when a segment isn't specified, or can place additional constraints on the value's that are valid for a given route parameter. This section takes a look at these features, and ways you can apply them.

5.4.1 Using optional and default values

In the previous section, you saw a simple route template with a literal segment and two required routing parameters. In figure 5.7, you can see a more complex route that uses a number of additional features.

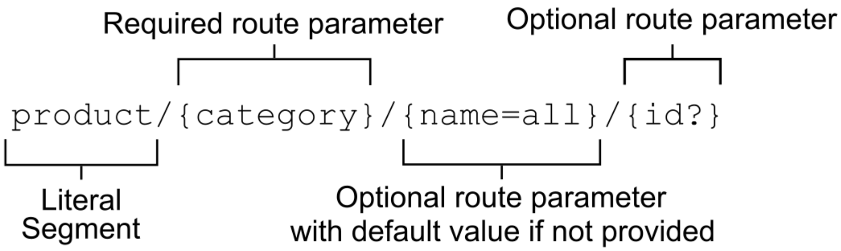

Required route parameter
Optional route parameter
product/{category}/{name=all}/{id?}
Literal Segment
Optional route parameter with default value if not provided

Figure 5.7 A more complex route template showing literal segments, named route parameters, optional parameters, and default values

The literal `product` segment and the required `{category}` parameter are the same as you saw in figure 5.6. The `{name}` parameter looks *similar*, but it has a default value specified for it using `=all`. If the URL doesn't contain a segment corresponding to the `{name}` parameter, then the router will use the `all` value instead.

The final segment of figure 5.7, {id?}, defines an optional route parameter called id. This segment of the URL is optional—if present, the router will capture the value for the {id} parameter; if it isn't there, then it won't create a route value for id.

You can specify any number of route parameters in your templates, and these values will be available to you when it comes to model binding. The complex route template of figure 5.7 allows you to match a greater variety of URLs by making {name} and {id} optional, and providing a default for {name}. Table 5.2 shows some of the possible URLs this template would match, and the corresponding route values the router would set.

Table 5.2 URLs that would match the template of figure 5.7 and their corresponding route values

URL	Route values
/product/shoes/formal/3	category=shoes, name=formal, id=3
/product/shoes/formal	category=shoes, name=formal
/product/shoes	category=shoes, name=all
/product/bags/satchels	category=bags, name=satchels
/product/phones	category=phones, name=all
/product/computers/laptops/ABC-123	category=computes, name=laptops, id=ABC-123

Note that there's no way to specify a value for the optional {id} parameter without also specifying the {category} and {name} parameters. You can *only* put an optional parameter (that doesn't have a default) at the *end* of a route template. For example, imagine your route template had an optional {category} parameter:

```
{category?}/{name}
```

Now try to think of a URL that would specify the {name} parameter, but not the {category}. It can't be done! Patterns like this won't cause errors per-se, the category parameter is just essentially required, even though you've marked is as optional.

Using default values allows you to have multiple ways to call the same URL, which may be desirable in some cases. For example, using the route template in figure 5.7 the two URLs below are equivalent:

- /product/shoes
- /product/shoes/all

Both of these URLs will execute the same Razor Page, with the same route values of category=shoes and name=all. Using default values allows you to use shorter and more memorable URLs in your application for common URLs, but still have the flexibility to match a variety of routes in a single template.

5.4.2 Adding additional constraints to route parameters

By defining whether a route parameter is required or optional, and whether it has a default value, you can match a broad range of URLs with a pretty terse template syntax. Unfortunately, in some cases this can end up being a little *too* broad. Routing only matches URL segments to route parameters; it doesn't know anything about the data that you're expecting those route parameters to contain. If you consider a template similar to that in figure 5.7, "`{category}/{name=all}/{id?}`, the following URLs would all match:

- `/shoes/sneakers/test`
- `/shoes/sneakers/123`
- `/Account/ChangePassword`
- `/ShoppingCart/Checkout/Start`
- `/1/2/3`

These URLs are all perfectly valid given the template's syntax, but some might cause problems for your application. These URLs all have two or three segments, and so the router happily assigns route values and matches the template when you might not want it to! The route values assigned are:

- `/shoes/sneakers/test—category=shoes, name=sneakers, id=test`
- `/shoes/sneakers/123—category=shoes, name=sneakers, id=123`
- `/Account/ChangePassword—category=Account, name=ChangePassword`
- `/Cart/Checkout/Start—category=Cart, name=Checkout, id=Start`
- `/1/2/3—category=1, name=2, id=3`

Typically, the router passes route values to Razor Pages through a process called model binding which you saw briefly in chapter 4 (and which we'll discuss in detail in the next chapter). For example, a Razor Page with the handler `public void OnGet(int id)` would obtain the `id` argument from the `id` route value. If the `id` route parameter ends up assigned a *non-integer* value from the URL, then you'll get an exception when it's bound to the *integer* `id` parameter.

To avoid this problem, it's possible to add additional *constraints* to a route template that must be satisfied for a URL to be considered a match. Constraints can be defined in a route template for a given route parameter using : (a colon). For example, `{id:int}` would add the `IntRouteConstraint` to the `id` parameter. For a given URL to be considered a match, the value assigned to the `id` route value must be convertible to an integer.

You can apply a large number of route constraints to route templates to ensure that route values are convertible to appropriate types. You can also check more advanced constraints, for example, that an integer value has a particular minimum value, or that a string value has a maximum length. Table 5.3 describes a number of the possible constraints available, but you can find a more complete list online at http://mng.bz/U11Q.

Table 5.3 A few route constraints and their behavior when applied

Constraint	Example	Match examples	Description
`int`	`{qty:int}`	`123, -123, 0`	Matches any `integer`
`Guid`	`{id:guid}`	`d071b70c-a812-4b54-87d2-7769528e2814`	Matches any `Guid`
`decimal`	`{cost:decimal}`	`29.99, 52, -1.01`	Matches any `decimal` value
`min(value)`	`{age:min(18)}`	`18, 20`	Matches `integer` values of 18 or greater
`length(value)`	`{name:length(6)}`	`andrew,123456`	Matches `string` values with a length of 6
`optional int`	`{qty:int?}`	`123, -123, 0, null`	Optionally matches any `integer`
`optional int max(value)`	`{qty:int:max(10)?}`	`3, -123, 0, null`	Optionally matches any `integer` of 10 or less

TIP As you can see from table 5.3, you can also combine multiple constraints; place a colon between them or use an optional mark (`?`) at the end.

Using constraints allows you to narrow down the URLs that a given route template will match. When the routing middleware matches a URL to a route template, it interrogates the constraints to check that they're all valid. If they aren't valid, then the route template isn't considered a match, and the Razor Page won't be executed.

WARNING Don't use route constraints to validate general input, for example to check that an email address is valid. Doing so will result in 404 "Page not found" errors, which will be confusing for the user.

Constraints are best used sparingly, but they can be useful when you have strict requirements on the URLs used by the application, as they can allow you to work around some otherwise tricky combinations.

Constraints and ordering in attribute routing

If you have a well-designed set of URLs for your application, then you will probably find that you don't really need to use route constraints. Route constraints really come in useful when you have "overlapping" route templates.

For example, imagine you have a Razor Page with the route template `"{number}/{name}"`, and another with the template `"{product}/{id}"`. When a request with the URL `/shoes/123` arrives, which template is chosen? They *both* match, so the routing middleware panics and throws an exception. Not ideal.

Using conventions can fix this. For example, if you update the first template to `"{number:int}/{name}"`, then the integer constraint means the URL is no longer a match, and the routing middleware can choose correctly. Note however, that the URL `/123/shoes` *does* still match both route templates, so you're not out of the woods.

Generally, you should avoid overlapping route templates like these, as they're often confusing and more trouble than they're worth.

Attribute routing (used by Razor Pages and MVC controllers for building APIs) allows you to explicitly control the *order* the routing middleware looks at your route templates, which can also be used to resolve issues like the one above. However, if you find yourself needing to manually control the order, this is a very strong indicator that your URLs are confusing.[17]

If your route templates are well defined such that each URL only maps to a single template, ASP.NET Core routing will work without any difficulties. Sticking to the built-in conventions as far as possible is the best way to stay on the happy path!

We're coming to the end of our look at route templates, but before we move on there's one more type of parameter to think about: the catch-all parameter.

5.4.3 Matching arbitrary URLs with the catch-all parameter

You've already seen how route templates take URL segments and attempt to match them to parameters or literal strings. These segments normally split around the slash character, `/`, so the route parameters themselves won't contain a slash. What do you do if you need them to contain a slash, or you don't know how many segments you're going to have?

Imagine you are building a currency-converter application that shows the exchange rate from one currency to one or more other currencies. You're told that the URLs for this page should contain all the currencies as separate segments. Here are some examples:

- `/USD/convert/GBP`—Show USD with exchange rate to GBP
- `/USD/convert/GBP/EUR`—Show USD with exchange rate to GBP and EUR
- `/USD/convert/GBP/EUR/CAD`—Show USD with exchange rate for GBP, EUR, and CAD

If you want to support showing *any* number of currencies as the URLs above do, then you need a way of capturing *everything* after the `convert` segment. You could achieve this for the

[17] If you're really sure you do need to control route template ordering, see the documentation at https://docs.microsoft.com/en-us/aspnet/core/razor-pages/razor-pages-conventions?route-order. Note that you can only control the order for additional routes added using conventions, as you'll see in section 5.7

Pages/Convert.cshtml Razor Page by using a catch-all parameter in the `@page` directive, as shown in figure 5.8.

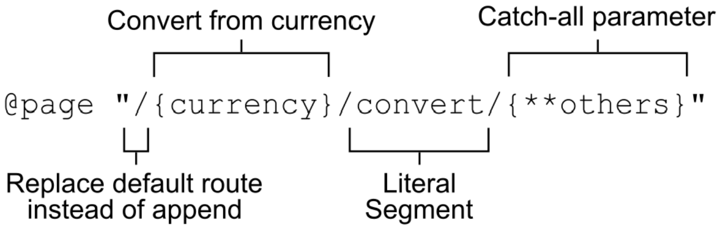

Convert from currency
Catch-all parameter
@page "/{currency}/convert/{**others}"
Replace default route instead of append
Literal Segment

Figure 5.8 You can use catch-all parameters to match the remainder of a URL. Catch-all parameters may include the "/" character or may be an empty string.

Catch-all parameters can be declared using either one or two asterisks inside the parameter definition, like `{*others}` or `{**others}`. These will match the remaining unmatched portion of a URL, including any slashes or other characters that aren't part of earlier parameters. They can also match an empty string. For the `USD/convert/GBP/EUR` URL, the value of the route value `others` would be the single string `"GBP/EUR"`.

> **TIP** Catch-all parameters are greedy and will capture the whole unmatched portion of a URL. Where possible, to avoid confusion, avoid defining route templates with catch-all parameters that "overlap" other route templates.

The one- and two-asterisk versions of the catch-all parameter behave identically when routing an incoming request to a Razor Page. The difference only comes when you're *generating* URLs (which we'll cover in the next section): the one-asterisk version URL encodes forward slashes, and the two-asterisk version doesn't. The "round-trip" behavior of the two-asterisk version is typically what you want.[18]

You read that correctly—mapping URLs to Razor Pages is only half of the responsibilities of the routing system in ASP.NET Core. It's also used to *generate* URLs so that you can easily reference your Razor Pages from other parts of your application.

[18] For details and examples of this behaviour, see the documentation https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing

5.5 Generating URLs from route parameters

In this section, we look at the other half of routing—generating URLs. You'll learn how to generate URLs as a string you can use in your code, and how to automatically send redirect URLs as a response from your Razor Pages.

One of the by-products of using the routing infrastructure in ASP.NET Core is that your URLs can be somewhat fluid. If you rename a Razor Page, then the URL associated with that page will also change. For example, renaming the Pages/Cart.cshtml page to Pages/Basket/View.cshtml would cause the URL used to access the page to change from `/Cart` to `/Basket/View`.

Trying to manually manage these links within your app would be a recipe for heartache, broken links, and 404s. If your URLs were hardcoded, then you'd have to remember to do a find-and-replace with every rename!

Luckily, you can use the routing infrastructure to generate appropriate URLs dynamically at runtime instead, freeing you from the burden. Conceptually, this is almost an exact reverse of the process of mapping a URL to a Razor Page, as shown in figure 5.9. In the "routing" case, the routing middleware takes a URL, matches it to a route template, and splits it into route values. In the "URL generation" case, the generator takes in the *route values* and combines them with a route template to build a URL.

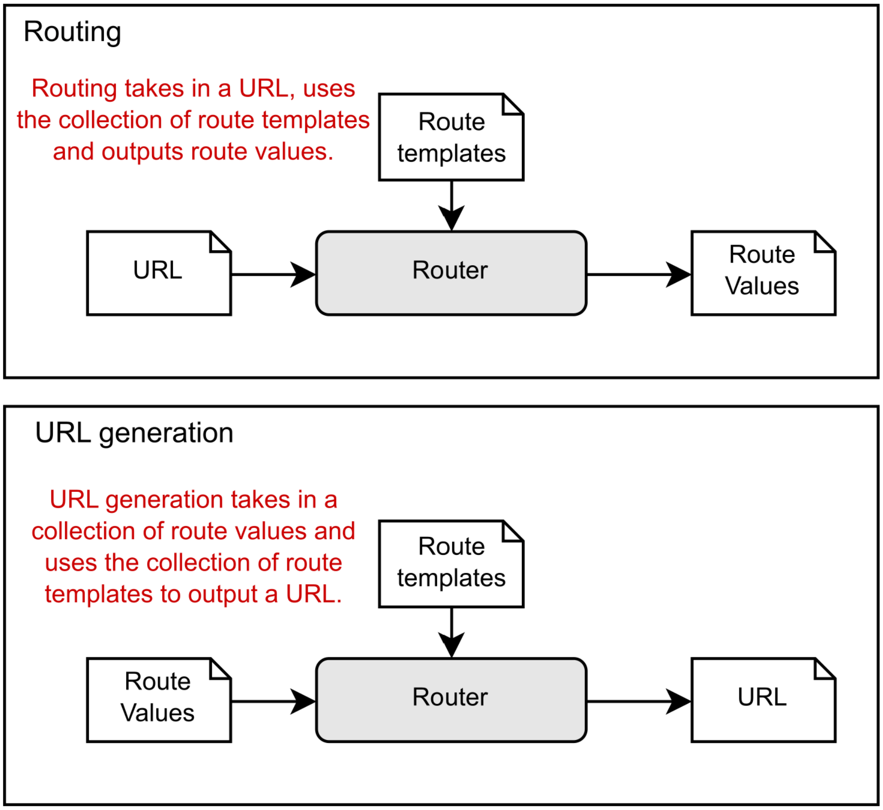

Routing
Routing takes in a URL, uses the collection of route templates and outputs route values.
Route templates
URL
Router
Route Values
URL generation
URL generation takes in a collection of route values and uses the collection of route templates to output a URL.
Route templates
Route Values
Router
URL

Figure 5.9 A comparison between routing and URL generation. Routing takes in a URL and generates route values, but URL generation uses route values to generate a URL.

5.5.1 Generating URLs for a Razor Page

You will need to generate URLs in various places in your application, and one common location is in your Razor Pages and MVC controllers. The following listing shows how you could generate a link to the Pages/Currency/View.cshtml Razor Page, using the `Url` helper from the `PageModel` base class.

Listing 5.2 Generating a URL using `IUrlHelper` and the Razor Page name

```
public class IndexModel : PageModel                                 #A
{
    public void OnGet()
    {
        var url = Url.Page("Currency/View", new { code = "USD" });  #B
    }
}
```

#A Deriving from PageModel gives access to the Url property.
#B You provide the relative path to the Razor Page, along with any additional route values.

The `Url` property is an instance of `IUrlHelper` that allows you to easily generate URLs for your application by referencing other Razor Pages by their file path. It exposes a `Page` method to which you pass the name of the Razor Page and any additional route data. The route data is packaged as key-value pairs into a single C# anonymous object. If you need to pass more than one route value, you can add additional properties to the anonymous object. The helper will then generate a URL based on the referenced page's route template.

> **TIP** You can provide the *relative* file path to the Razor Page, as shown in Listing 5.2. Alternatively, you can provide the *absolute* file path (relative to the Pages folder) by starting the path with a `"/"`, for example `"/Currency/View"`.

The `IUrlHelper` has several different overloads of the `Page` method. Some of these methods allow you to specify a specific page handler, others let you generate an absolute URL instead of a relative URL, and some let you pass in additional route values.

In listing 5.2, as well as providing the file path, I passed in an anonymous object, `new { code = "USD" }`. This object provides additional route values when generating the URL, in this case setting the `code` parameter to `"USD"`.

If a selected route explicitly includes the defined route value in its definition, such as in the `"Currency/View/{code}"` route template, then the route value will be used in the URL path, giving `/Currency/View/GBP`.

If a route doesn't contain the route value explicitly, for example in the `"Currency/View"` template, then the route value is appended as additional data as part of the query string, for example `/Currency/View?code=GBP`.

Generating URLs based on the page you want to execute is convenient, and the usual approach taken in most cases. If you're using MVC controllers for your APIs then the process is much the same as for Razor Pages, though the methods are slightly differently.

5.5.2 Generating URLs for an MVC controller

Generating URLs for MVC controllers is very similar to Razor Pages. The main difference is that you use the `Action` method on the `IUrlHelper`, and you provide an MVC controller name and action name instead of a page path. The following listing shows an MVC controller generating a link from one action method to another, using the `Url` helper from the `Controller` base class.

Listing 5.3 Generating a URL using `IUrlHelper` and the action name

```
public class CurrencyController : Controller                        #A
{
    [HttpGet("currency/index")]
    public IActionResult Index()
    {
        var url = Url.Action("View", "Currency",                       #B
            new { code = "USD" });                                     #B
        return Content(${"The URL is {url}");                    #C
    }

    [HttpGet("currency/view/{code}")]
    public IActionResult View(string code)                       #D
    {
         /* method implementation*/
    }
}
```

#A Deriving from Controller gives access to the Url property.
#B You provide the controller and action name, along with any additional route values.
#C This will return "The URL is /Currency/View/USD".
#D The URL generated will route to the View action method.

You can call the `Action` and `Page` methods on `IUrlHelper` from both Razor Pages and MVC controllers, so you can generate links back and forth between them if you need to. The important question is: what is the *destination* of the URL? If the URL you need refers to a Razor Page, use the `Page` method. If the destination is an MVC action, use the `Action` method.

> **TIP** **Instead of using strings for the name of the action method, use the C# 6** `nameof` **operator to make the value refactor-safe, for example,** `nameof(View)`**.**

If you're routing to an action in the same controller, you can use a different overload of `Action` that omits the controller name when generating the URL. The `IUrlHelper` uses *ambient values* from the current request and overrides these with any specific values you provide.

> **DEFINITION** **Ambient values are the route values for the current request. They include** `controller` **and** `action` **when called from an MVC controller but can also include additional route values that were set when the action or Razor Page was initially located using routing. See the documentation for further details: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing.**

Generating URLs using the `Url` property doesn't tend to be very common in practice. Instead, it's more common to generate URLs implicitly with an `ActionResult`.

5.5.3 Generating URLs with ActionResults

In this section you've seen how to generate a string containing a URLs for both Razor Pages and MVC actions. This is useful if you need to display the URL to a user, or to include the URL in an API response for example. However, you don't need to display URLs very often. More commonly, you want to *automatically redirect* a user to a URL. For that situation you can use an `ActionResult` to handle the URL generation instead.

The listing below shows how you can generate a URL that automatically redirects a user to a different Razor Pages using an `ActionResult`. The `RedirectToPage` method takes the path to a Razor Page and any required route parameters, and generates a URL in the same way as the Url.Page method. The framework automatically sends the generated URL as the response, so you never "see" the URL in your code. The user's browser then reads the URL from the response, and automatically redirects to the new page.

Listing 5.4 Generating a redirect URL from an ActionResult

```
public class CurrencyModel : PageModel
{
    public IActionResult OnGetRedirectToPage()
    {
        return RedirectToPage("Currency/View", new { id = 5 });  #A
    }
}
```

#A The RedirectToPage method generates a RedirectToPageResult with the generated URL.

You can use a similar method, RedirectToAction, to automatically redirect to an MVC action instead. Just as with the Page and Action methods, it is the *destination* that controls whether you need to use RedirectToPage or RedirectToAction. RedirectToAction is only necessary if you're using MVC controllers to generate HTML instead of Razor Pages.

> **TIP I recommend you use Razor Pages instead of MVC controllers for HTML generation. For a discussion on the benefits of Razor Pages, refer to chapter 4.**

As well as generating URLs from your Razor Pages and MVC controllers, you'll often find you need to generate URLs when building HTML in your views. This is necessary in order to provide navigation links in your web application. You'll see how to achieve this when we look at Razor Tag Helpers in chapter 8.

If you need to generate URLs from parts of your application outside of the Razor Page or MVC infrastructure, then you won't be able to use the IUrlHelper helper or an ActionResult. Instead, you can use the LinkGenerator class.

5.5.4 Generating URLs from other parts of your application

If you're writing your Razor Pages and MVC controllers following the advice from chapter 4, then you should be trying to keep your Razor Pages relatively simple. That requires you to execute your application's business and domain logic in separate classes and services.

For the most part, the URLs your application uses *shouldn't* be part of your domain logic. That makes it easier for your application to evolve over time, or even change completely. For example, you may want to create a mobile application that reuses the same business logic for an ASP.NET Core app. In that case, using URLs in the business logic wouldn't make sense, as they wouldn't be correct when the logic is called from the mobile app!

> **TIP** Where possible, try to keep knowledge of the "front-end" application design out of your business logic. This pattern is known generally as the Dependency Inversion principle.[19]

Unfortunately, sometimes that separation is not possible, or it makes things significantly more complicated. One example might be when you're creating emails in a background service—it's likely you'll need to include a link to your application in the email. The `LinkGenerator` class lets you generate that URL, so that it updates automatically if the routes in your application change.

The `LinkGenerator` class is available in any part of your application, so you can use it inside middleware and any other services. You can use it from Razor Pages and MVC too if you wish, but the `IUrlHelper` available is typically easier and hides some details of using the `LinkGenerator`.

`LinkGenerator` has various methods for generating URLs, such as `GetPathByPage`, `GetPathByAction`, and `GetUriByPage` as shown in the listing below. There are some subtleties to using these methods, especially in complex middleware pipelines, so stick to the `IUrlHelper` where possible, and be sure to consult the documentation[20] if you have problems.

Listing 5.5 Generating URLs using the LinkGeneratorClass

```
public class CurrencyModel : PageModel
{
    private readonly LinkGenerator _link;                         #A
    public CurrencyModel(LinkGenerator linkGenerator)             #A
    {                                                             #A
        _link = linkGenerator;                                    #A
    }                                                             #A

    public void OnGet ()
    {
        var url1 = Url.Page("Currency/View", new { id = 5 });     #B
        var url3 = _link.GetPathByPage(                           #C
                       HttpContext,                               #C
                       "/Currency/View",                          #C
                       values: new { id = 5 });                   #C
        var url2= _link.GetPathByPage(                            #D
                      "/Currency/View",                           #D
                      values: new { id = 5 });                    #D
        var url4 = _link.GetUriByPage(                            #E
                       page: "/Currency/View",                    #E
                       handler: null,                             #E
                       values: new { id = 5 },                    #E
```

[19] Steve Smith maintains a project template that applies this pattern rigorously, called the Clean Architecture. You can find the template, along with many useful links and books on the topic here: https://github.com/ardalis/CleanArchitecture.

[20] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing#url-generation-with-linkgenerator

```
                       scheme: "https",                          #E
                       host: new HostString("example.com"));     #E
    }
}
```

#A LinkGenerator can be accessed using dependency injection
#B Url can generate relative paths to using Url.Page. You can use relative or absolute Page paths.
#C GetPathByPage is equivalent to Url.Page when you pass in HttpContext. Can't use relative paths
#D Other overloads don't require an HttpContext
#E GetUriByPage generates an absolute URL instead of a relative URL

Whether you're generating URLs using the `IUrlHelper` or `LinkGenerator` you need to be careful when using the route generation methods. Make sure you provide the correct Razor Page path, and any necessary route parameters. If you get something wrong—have a type in your path or forgot to include a required route parameter, for example—then the URL generated will be `null`. It's worth checking the generated URL for `null` explicitly, just to be sure there's not problems!

So far in this chapter we've looked extensively at how incoming requests are routed to Razor Pages, but we haven't really seen where page handlers come into it. In the next section we look at page handlers and how your can have multiple handlers on a Razor Page.

5.6 Selecting a page handler to invoke

At the start of this chapter I said routing was about mapping URLs to a *handler*. For Razor Pages, that means a *page handler*, but so far, we've only been talking about routing based on a Razor Page's route template. In this section you'll learn how the `EndpointMiddleware` selects which page handler to invoke when it executes a Razor Page.

You learned about page handlers in chapter 4, and their role within Razor Pages, but we haven't discussed how a page handler is *selected* for a given request. Razor Pages can have multiple handlers, so if the `RoutingMiddleware` selects a Razor Page, the `EndpointMiddleware` still needs to know a way to choose which handler to execute.

Consider the Razor Page `PageModel` shown in the listing below. This Razor Page has three handlers `OnGet`, `OnPostAsync`, and `OnPostCustomSearch`. The body of the handler methods aren't shown, as, at this point, we're only interested in how the `RoutingMiddleware` chooses which handler to invoke.

Listing 5.6 Razor Page with multiple page handlers

```
public class SearchModel : PageModel
{
    public void OnGet()                  #A
    {
        // Handler implementation
    }

    public Task OnPostAsync()             #B
    {
        // Handler implementation
    }
    public void OnPostCustomSearch()      #C
    {
        // Handler implementation
    }
}
```

#A Handles GET requests.
#B Handles POST requests. The async suffix is optional, and is ignored for routing purposes.
#C Handles POST requests where the handler route value has the value CustomSearch

Razor Pages can contain any number of page handlers, but only one runs in response to a given request. When the `EndpointMiddleware` executes a selected Razor Page, it selects a page handler to invoke based on two variables:

- The HTTP verb used in the request (for example `GET`, `POST`, or `DELETE`)
- The value of the `handler` route value.

The `handler` route value typically comes from a query string value in the request URL, for example `/Search?handler=CustomSearch`. If you don't like the look of query strings (I don't!) you can include the `{handler}` route parameter in your Razor Page's route template. For example, for the Search page in listing 5.6, you could update the page's directive to

```
@page "{handler}"
```

This would give a complete route template something like `"Search/{handler}"`, which would match URLs such as `/Search/CustomSearch`.

The `EndpointMiddleware` uses the handler route value and the HTTP verb together with a standard naming convention to identify which page handler to execute, as shown in figure 5.10. The handler parameter is optional, and typically provided as part of the request's query string or as a route parameter, as described above. The async suffix is also optional and is often used when the handler uses asynchronous programming constructs such as `Task` or `async`/`await`[21].

[21] The async suffix naming convention is suggested by Microsoft, though it is unpopular with some developers. NServiceBus provide a reasoned argument against it here (along with Microsoft's advice): https://docs.particular.net/nservicebus/upgrades/5to6/async-suffix.

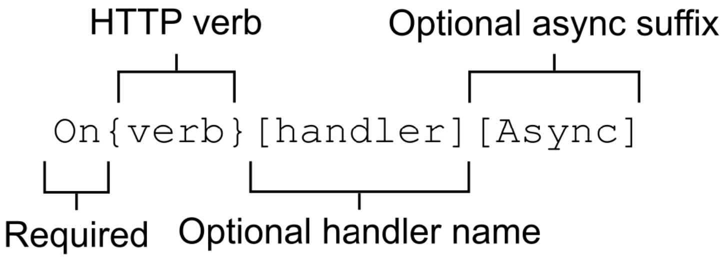


Figure 5.10 Razor Page handlers are matched to a request based on the HTTP verb and the optional handler parameter.

Based on this convention, we can now identify what type of request each page handler in listing 5.6 corresponds to:

- `OnGet`—Invoked for `GET` requests that don't specify a `handler` value.
- `OnPostAsync`—Invoked for `POST` requests that don't specify a `handler` value. Returns a `Task` so uses the `Async` suffix, which is ignored for routing purposes.
- `OnPostCustomSearch`—Invoked for `POST` requests that specify a `handler` value of `"CustomSearch"`.

The Razor Page in listing 5.6 specifies three handlers, so it can handle only three verb-handler pairs. But what happens if you get a request that doesn't match these? Such as a request using the `DELETE` verb, a `GET` request with a non-blank `handler` value, or a `POST` request with an unrecognized `handler` value?

For all these cases, the `EndpointMiddleware` executes an *implicit* page handler instead. Implicit page handlers contain no logic, they just render the Razor view. For example, if you sent a `DELETE` request to the Razor Page in listing 5.6, an implicit handler would be executed. The implicit page handler is equivalent to the following handler code:

```
public void OnDelete() { }
```

DEFINITION **If a page handler does not match a request's HTTP verb and handler value, an *implicit* page handler is executed which renders the associated Razor view. Implicit page handlers take part in model binding and use page filter but execute no logic.**

There's one exception to the implicit page handler rule: if a request uses the `HEAD` verb, and there is no corresponding `OnHead` handler, Razor Pages will execute the `OnGet` handler instead (if it exists).[22]

[22] HEAD requests are typically sent automatically by the browser, and don't return a response body. They're often used for security purposes, as you'll see in chapter 18.

At this point we’ve covered mapping request URLs to Razor Pages, and generating URLs using the routing infrastructure, but most of the URLs we’ve been using have been kind of ugly. If seeing capital letters in your URLs bothers you, then the next section is for you. In the next section we customize the conventions your application uses to generate route templates.

5.7 Customizing conventions with Razor Pages

Razor Pages is built on a series of conventions that are designed to reduce the amount of boilerplate code you need to write. In this section you’ll see some of the ways you can customize those conventions. By customizing the conventions Razor Pages uses in your application you get full control over your application’s URLs without having to manually customize every Razor Page’s route template.

By default, ASP.NET Core generates URLs that match the file names of your Razor Pages very closely. For example, the Razor Page located at the path Pages/Products/ProductDetails.cshtml, would correspond to the route template `Products/ProductDetails`.

These days, it’s not very common to see capital letters in URLs. Similarly, words in URLs are usually separated using “kebab-case” rather than “PascalCase”, for example `product-details` instead of `ProductDetails`. Finally, it’s also common to ensure your URLs always end with a trailing slash, for example `/product-details/` instead of `/product-details`. Razor Pages gives you complete control over the conventions your application uses to generate route templates, but these are two common changes I make.

The following listing shows how to ensure URLs are always lowercase, and that they always have a trailing slash. You can change these conventions by configuring a `RouteOptions` object in Startup.cs. This object contains configuration for the whole ASP.NET Core routing infrastructure, so any changes you make will apply to both Razor Pages and MVC. You’ll learn more about configuring options in chapter 10.

Listing 5.7 Configuring routing conventions using `RouteOptions` in Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();                            #A
    services.Configure<RouteOptions>(options =>          #B
    {
        options.AppendTrailingSlash = true;              #C
        options.LowercaseUrls = true;                    #C
        options.LowercaseQueryStrings = true;            #C
    });
}
```

#A Add the standard Razor Pages services
#B Configure the RouteOptions object after by providing a configuration method
#C You can change the conventions used to generate URLs. By default, these properties are false.

To use kebab-case for your application, annoyingly you must create a custom parameter transformer. This is a somewhat advanced topic, but it’s relatively simple to implement in this case. The listing below shows how you can create a parameter transformer that uses a Regular Expression to replace PascalCase values in a generated URL with kebab-case

Listing 5.8 Creating a kebab-case parameter transformer

```
public class KebabCaseParameterTransformer                   #A
    : IOutboundParameterTransformer                          #A
{
    public string TransformOutbound(object value)
    {
        if (value == null) return null;                       #B

        return Regex.Replace(value.ToString(),                #C
            "([a-z])([A-Z])", "$1-$2").ToLower();             #C
    }
}
```

#A Create a class that implements the parameter transformer interface
#B Guard against null values to avoid runtime exceptions
#C The regular expression replaces PascalCase patterns with kebab-case

You can register the parameter transformer in your application with the `AddRazorPagesOptions` extension method in Startup.cs. This method is chained after the `AddRazorPages` method and can be used to completely customize the conventions used by Razor Pages. The listing below shows how to register the kebab-case transformer. It also shows how to add an extra page route convention for a given Razor Page.

Listing 5.9 Registering a parameter transformer using `RazorPagesOptions`

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddRazorPagesOptions(opts =>                                 #A
        {
            opts.Conventions.Add(                                     #B
                new PageRouteTransformerConvention(                   #B
                    new KebabCaseParameterTransformer()));            #B
            opts.Conventions.AddPageRoute(                            #C
                "/Search/Products/StartSearch", "/search-products");    #C
        });
}
```

#A AddRazorPagesOptions can be used to customize the conventions used by Razor Pages.
#B Registers the parameter transformer as a convention used by all Razor Pages.
#C AddPageRoute adds an additional route template to Pages//Search/Products/StartSearch.cshtml.

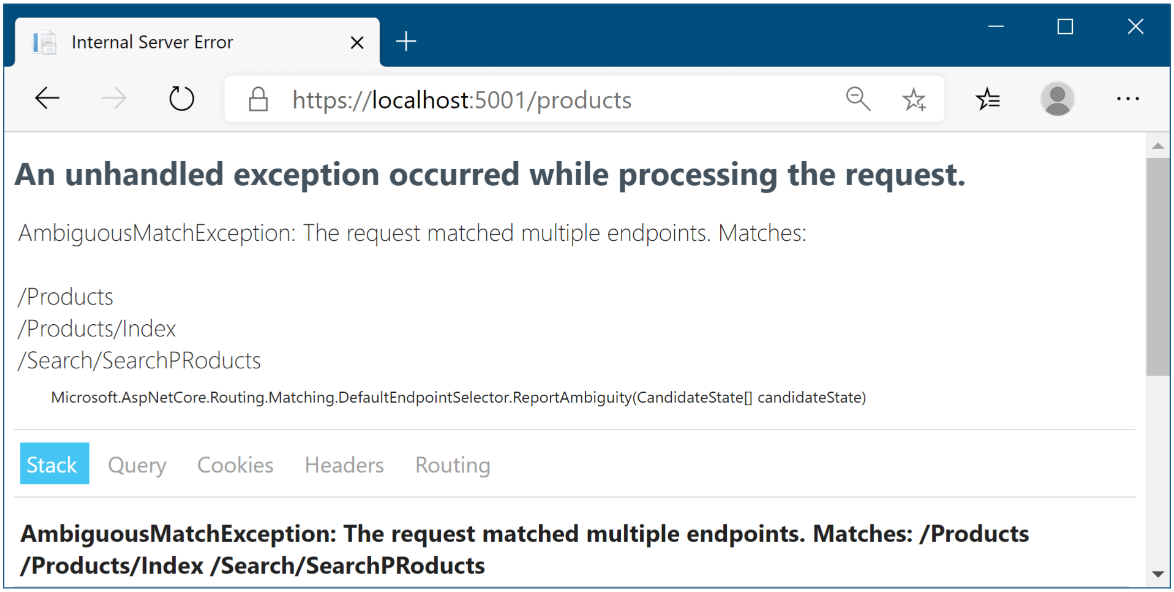
The `AddPageRoute` convention adds an alternative way to execute a single Razor Page. Unlike when you customize the route template for a Razor Page using the `@page` directive, using `AddPageRoute` *adds an extra route template* to the page instead of replacing the default. That means there are two route templates that can access the page.

There are many other ways you can customize the conventions for your Razor Pages applications, but most of the time that's not necessary. If you do find you need to customize all the pages in your application in some way, for example to add an extra header to every page's response, you can use a custom convention. The documentation contains details on everything that's available: https://docs.microsoft.com/en-us/aspnet/core/razor-pages/razor-pages-conventions.

Conventions are a key feature of Razor Pages and you should lean on them whenever you can. While you *can* manually override the route templates for individual Razor Pages, as you've seen in previous sections, I advise against it where possible. In particular:

- **Avoid** replacing the route template with an absolute path in a page's `@page` directive.
- **Avoid** adding literal segments to the `@page` directive. Rely on the file hierarchy instead.
- **Avoid** adding additional route templates to a Razor Page with the `AddPageRoute` convention. Having multiple ways to access a page can sometimes be confusing.
- **Do** add route parameters to the `@page` directive to make your routes dynamic, for example `@page {name}`.
- **Do** consider using global conventions when you want to change the route templates for all your Razor Pages, for example using kebab-case as you saw in the previous section.

In a nutshell, these rules amount to "stick to the conventions". The danger, if you don't, is that you accidentally create two Razor Pages that have overlapping route templates. Unfortunately, if you end up in that situation, you won't get an error at compile time. Instead, you'll get an exception at runtime when your application receives a request that matches multiple route template, as shown in figure 5.11.



An unhandled exception occurred while processing the request.

AmbiguousMatchException: The request matched multiple endpoints. Matches:

/Products
/Products/Index
/Search/SearchPRoducts

Microsoft.AspNetCore.Routing.Matching.DefaultEndpointSelector.ReportAmbiguity(CandidateState[] candidateState)

AmbiguousMatchException: The request matched multiple endpoints. Matches: /Products /Products/Index /Search/SearchPRoducts

Figure 5.11 If multiple Razor Pages are registered with overlapping route templates, you'll get an exception at runtime when the router can't work out which one to select.

Congratulations, you've made it all the way through this detailed discussion on routing! Routing is one of those topics that people often get stuck on when they come to building an application, which can be frustrating. You'll revisit routing again when I describe how to create Web APIs in chapter 9, but rest assured, you've already covered all the tricky details in this chapter!

In chapter 6, we'll dive into model binding. You'll see how the route values generated during routing are bound to your action method parameters, and perhaps more importantly, how to validate the values you're provided.

5.8 Summary

- Routing is the process of mapping an incoming request URL to a Razor Page that will execute to generate a response. You can use routing to decouple your URLs from the files in your project and to have multiple URLs map to the same Razor Page.
- ASP.NET Core uses two pieces of middleware for routing. The `EndpointRoutingMiddleware` is added in Startup.cs by calling `UseRouting()` and the `EndpointMiddleware` is added by calling `UseEndpoints()`.
- The `EndpointRoutingMidleware` selects which endpoint should be executed by using routing to match the request URL. The `EndpointMiddleware` executes the endpoint.
- Any middleware placed between the calls to `UseRouting()` and `UseEndpoints()` can tell which endpoint will be executed for the request.
- Route templates define the structure of known URLs in your application. They're strings with placeholders for variables that can contain optional values and map to Razor Pages or to MVC controller actions.
- Route parameters are variable values extracted from a request's URL.
- Route parameters can be optional and can have default values used when they're missing.
- Route parameters can have constraints that restrict the possible values allowed. If a route parameter doesn't match its constraints, the route isn't considered a match.
- Don't use route constraints as general input validators. Use them to disambiguate between two similar routes.
- Use a catch-all parameter to capture the remainder of a URL into a route value.
- You can use the routing infrastructure to generate internal URLs for your application.
- The `IUrlHelper` can be used to generate URLs as a string based on an action name or Razor Page.
- You can use the `RedirectToAction` and `RedirectToPage` methods to generate URLs while also generating a redirect response.
- The `LinkGenerator` can be used to generate URLs from other services in your application, where you don't have access to an `HttpContext` object.
- When a Razor Page is executed, a single page handler is invoked based on the HTTP verb of the request, and the value of the `handler` route value.

- If there is no page handler for a request, an implicit page handler is used that renders the Razor view.
- You can control the routing conventions used by ASP.NET Core by configuring the `RouteOptions` object, for example to force all URLs to be lowercase, or to always append a trailing slash.
- You can add additional routing conventions for Razor Pages by calling `AddRazorPagesOptions()` after `AddRazorPages()` in Startup.cs. These conventions can control how route parameters are displayed or can add additional route templates for specific Razor Pages.
- Where possible, avoid customizing the route templates for a Razor Page and rely on the conventions instead.

6

The binding model: retrieving and validating user input

This chapter covers

- **Using request values to create binding models**
- **Customizing the model binding process**
- **Validating user input using `DataAnnotations` attributes**

In chapter 5, I showed you how to define a route with parameters—perhaps for the day in a calendar or the unique ID for a product page. But say a user requests a given product page—what then? Similarly, what if the request includes data from a form, to change the name of the product, for example? How do you handle that request and access the values the user provided?

In the first half of this chapter, we look at using *binding models* to retrieve those parameters from the request so that you can use them in your Razor Pages. You'll see how to take the data posted in the form or in the URL and *bind* them to C# objects. These objects are passed to your Razor Page handlers as method parameters or are set as properties on your Razor Page `PageModel`. When your page handler executes, it can use these values to do something useful—return the correct diary entry or change a product's name, for instance.

Once your code is executing in a page handler method, you might be forgiven for thinking that you can happily use the binding model without any further thought. Hold on now, where did that data come from? From a user—you know they can't be trusted! The second half of the chapter focuses on how to make sure that the values provided by the user are valid and make sense for your app.

You can think of the binding models as the *input* to a Razor Page, taking the user's raw HTTP request and making it available to your code by populating "plain old CLR objects"

(POCOs). Once your page handler has run, you're all set up to use the *output* models in ASP.NET Core's implementation of MVC—the view models and API models. These are used to generate a response to the user's request. We'll cover them in chapters 7 and 9.

Before we go any further, let's recap the MVC design pattern and how binding models fit into ASP.NET Core.

6.1 Understanding the models in Razor Pages and MVC

In this section I describe how *binding models* fit into the MVC design pattern we covered in chapter 4. I describe the difference between binding models and the other "model" concepts in the MVC pattern, and how they're each used in ASP.NET Core.

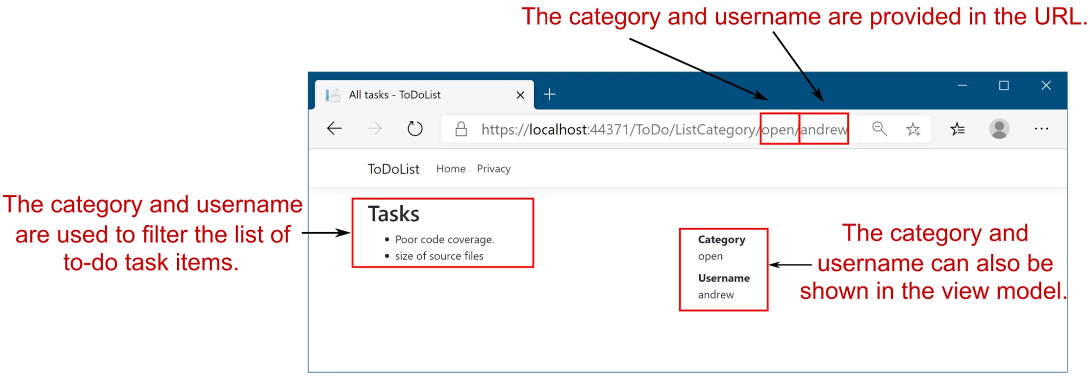
MVC is all about the separation of concerns. The premise is that by isolating each aspect of your application to focus on a single responsibility, it reduces the interdependencies in your system. This separation makes it easier to make changes without affecting other parts of your application.

The classic MVC design pattern has three independent components:

- *Controller*—Calls methods on the model and selects a view.
- *View*—Displays a representation of data that makes up the model.
- *Model*—The data to display and the methods for updating itself.

In this representation, there's only one model, the application model, which represents all the business logic for the application, as well as how to update and modify its internal state. ASP.NET Core has multiple models, which takes the single responsibility principle one step further than some views of MVC.

In chapter 4, we looked at an example of a to-do list application that can show all the to-do items for a given category and username. With this application, you make a request to a URL that's routed using `todo/listcategory/{category}/{username}`. This returns a response showing all the relevant to-do items, as shown in figure 6.1.



The category and username are provided in the URL.
All tasks - ToDoList
https://localhost:44371/ToDo/ListCategory/open/andrew
ToDoList
Home
Privacy
The category and username are used to filter the list of to-do task items.
Tasks
Poor code coverage.
size of source files
Category
open
Username
andrew
The category and username can also be shown in the view model.

Figure 6.1 A basic to-do list application that displays to-do list items. A user can filter the list of items by changing the `category` and `username` parameters in the URL.

The application uses the same MVC constructs you've already seen, such as routing to a Razor Page handler, as well as a number of different *models*. Figure 6.2 shows how a request to this application maps to the MVC design pattern and how it generates the final response, including additional details around the model binding and validation of the request.

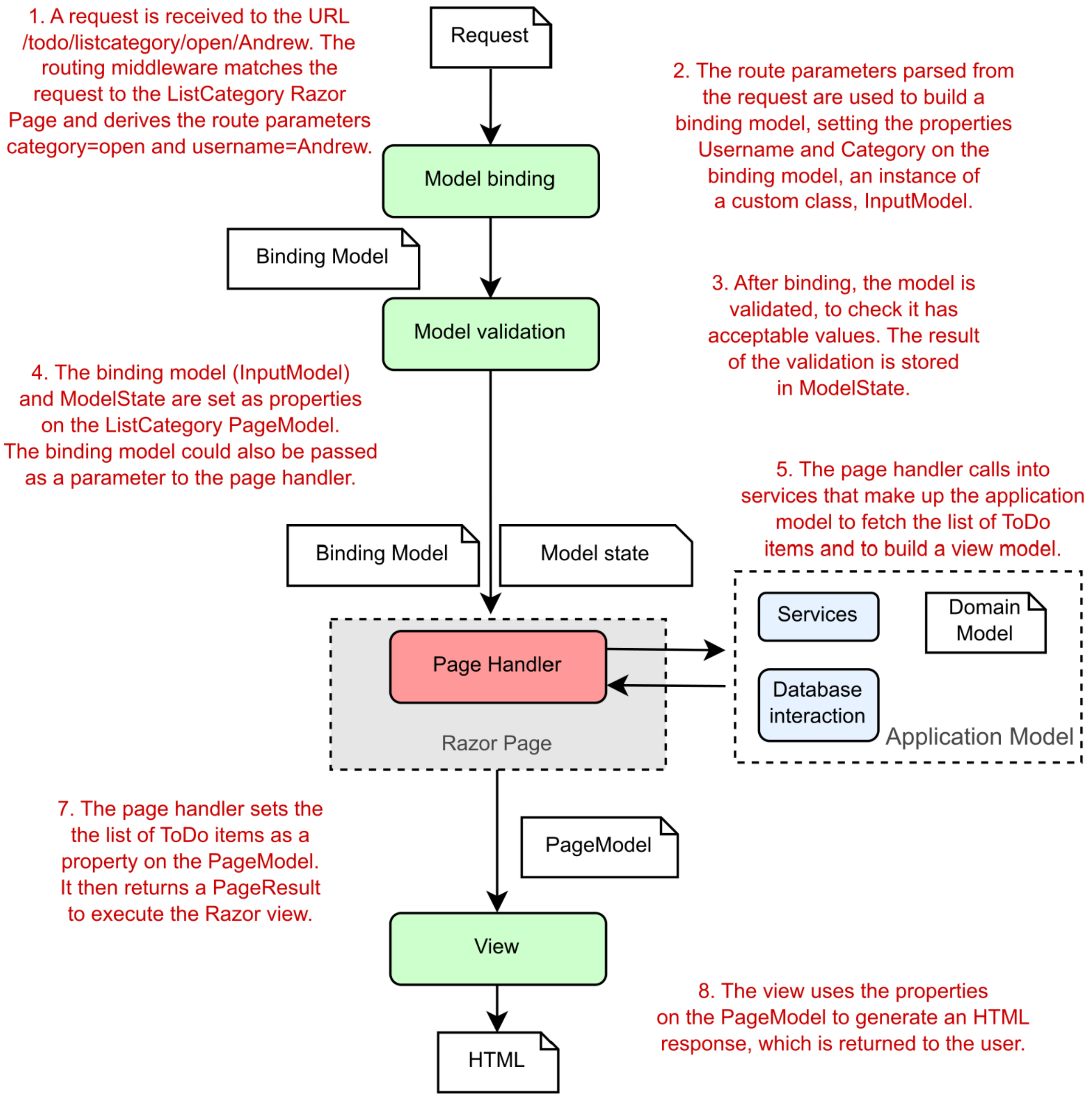

1. A request is received to the URL /todo/listcategory/open/Andrew. The routing middleware matches the request to the ListCategory Razor Page and derives the route parameters category=open and username=Andrew.
Request
2. The route parameters parsed from the request are used to build a binding model, setting the properties Username and Category on the binding model, an instance of a custom class, InputModel.
Model binding
Binding Model
3. After binding, the model is validated, to check it has acceptable values. The result of the validation is stored in ModelState.
Model validation
4. The binding model (InputModel) and ModelState are set as properties on the ListCategory PageModel. The binding model could also be passed as a parameter to the page handler.
5. The page handler calls into services that make up the application model to fetch the list of ToDo items and to build a view model.
Binding Model
Model state
Services
Domain Model
Page Handler
Database interaction
Razor Page
Application Model
7. The page handler sets the the list of ToDo items as a property on the PageModel. It then returns a PageResult to execute the Razor view.
PageModel
View
8. The view uses the properties on the PageModel to generate an HTML response, which is returned to the user.
HTML

Figure 6.2 The MVC pattern in ASP.NET Core handling a request to view a subset of items in a to-do list Razor Pages application.

ASP.NET Core Razor Pages uses several different models, most of which are POCOs, and the application model, which is more of a concept around a collection of services. Each of the models in ASP.NET Core is responsible for handling a different aspect of the overall request:

- *Binding model*—The binding model is all the information that's provided by the user when making a request, as well as additional contextual data. This includes things like route parameters parsed from the URL, the query string, and form or JSON data in the request body. The binding model itself is one or more POCO objects that you define. Binding models in Razor Pages are typically defined by creating a public property on the page's `PageModel` and decorating it with the `[BindProperty]` attribute. They can also be passed to a page handler as parameters.

  For this example, the binding model would include the name of the category, `open`, and the username, `Andrew`. The Razor Pages infrastructure inspects the binding model before the page handler executes to check whether the provided values are valid, though the page handler will execute even if they're not, as you'll see when we discuss validation in section 6.3

- *Application model*—The application model isn't really an "ASP.NET Core model" at all. It's typically a whole group of different services and classes, and is more of a "concept"—anything needed to perform some sort of business action in your application. It may include the domain model (which represents the thing your app is trying to describe) and database models (which represent the data stored in a database), as well as any other, additional services.

  In the to-do list application, the application model would contain the complete list of to-do items, probably stored in a database, and would know how to find only those to-do items in the `open` category assigned to `Andrew`.

  Modeling your domain is a huge topic, with many different possible approaches, so it's outside the scope of this book, but we'll touch briefly on creating database models in chapter 11.

- *Page model—The* `PageModel` *of a Razor Page serves two main functions: it acts as the controller for the application by exposing page handler methods, and it acts as the view model for a Razor view. All the data* required for the view to generate a response is exposed on the `PageModel`, such as the list of to-dos in the `open` category assigned to `Andrew`*.*

  The `PageModel` base class that you derive your Razor Pages from contains various helper properties and methods. One of these, the `ModelState` property, contains the result of the model validation as a series of key-value pairs. You'll learn more about validation and the `ModelState` property in section 6.3.

These models make up the bulk of any Razor Pages application, handling the input, business logic, and output of each page handler. Imagine you have an e-commerce application that allows users to search for clothes by sending requests to the URL `/search/{query}` URL, where `{query}` holds their search term:

- *Binding model*—Would take the `{query}` route parameter from the URL and any values posted in the request body (maybe a sort order, or number of items to show), and bind them to a C# class, which typically acts as a "throwaway" data transport class. This would be set as a property on the `PageModel` when the page handler is invoked.

- *Application model*—The services and classes that perform the logic. When invoked by the page handler, this would load all the clothes that match the query, applying the necessary sorting and filters, and return the results back to the controller.
- *Page model*—The values provided by the application model would be set as properties on the Razor Page's `PageModel`, along with other metadata, such as the total number of items available, or whether the user can currently check out. The Razor view would use this data to render the Razor view to HTML.

The important point about all these models is that their responsibilities are well-defined and distinct. Keeping them separate and avoiding reuse helps to ensure your application stays agile and easy to update.

The obvious exception to this separation is the `PageModel`, as it is where the binding models are defined, where the page handlers are defined, and it also holds the data required for rendering the view. Some people may consider the apparent lack of separation to be sacrilege, but in reality, it's not generally an issue. The lines of demarcation are pretty apparent, so as long as you don't try to, for example, invoke a page handler from inside a Razor view then you shouldn't run into any problems!

Now that you've been properly introduced to the various models in ASP.NET Core, it's time to focus on how to use them. This chapter looks at the binding models that are built from incoming requests—how are they created, and where do the values come from?

6.2 From request to model: making the request useful

In this section you will learn

- How ASP.NET Core creates binding models from a request.
- How to bind simple types, like `int` and `string`, as well as complex classes.
- How to choose which parts of a request are used in the binding model.

By now, you should be familiar with how ASP.NET Core handles a request by executing a page handler on a Razor Page. You've also already seen several page handlers, such as

```
public void OnPost(ProductModel product)
```

Page handlers are normal C# methods, so the ASP.NET Core framework needs to be able to call them in the usual way. When page handlers accept parameters as part of their method signature, such as `product` in the preceding example, the framework needs a way to generate those objects. Where exactly do they come from, and how are they created?

I've already hinted that in most cases, these values come from the request itself. But the HTTP request that the server receives is a series of strings—how does ASP.NET Core turn that into a .NET object? This is where *model binding* comes in.

DEFINITION *Model binding* extracts values from a request and uses them to create .NET objects. These objects are passed as method parameters to the page handler being executed or are set as properties of the `PageModel` that are marked with the `[BindProperty]` attribute.

The model binder is responsible for looking through the request that comes in and finding values to use. It then creates objects of the appropriate type and assigns these values to your model in a process called *binding*.

NOTE Model binding in Razor Pages and MVC is a one-way population of objects from the request, not the two-way databinding that desktop or mobile development sometimes uses.

Any properties on your Razor Page's `PageModel` (in the cshtml.cs file for your Razor Page), that are decorated with the `[BindProperty]` attribute, are created from the incoming request using model binding, as shown in the listing below. Similarly, if your page handler method has any parameters, these are also created using model binding.

Listing 6.1 Model binding requests to properties in a Razor Page

```
public class IndexModel: PageModel
{
    [BindProperty]                                 #A
    public string Category { get; set; }           #A

    [BindProperty(SupportsGet = true)]             #B
    public string Username { get; set; }           #B

    public void OnGet()
    {
    }

    public void OnPost(ProductModel model)     #C
    {
    }
}
```

#A Properties decorated with [BindProperty] take part in model binding
#B Properties are not model bound for GET requests, unless you use SupportsGet
#C Parameters to page handlers are also model bound when that handler is selected

As shown in the previous listing, `PageModel` properties are *not* model bound for `GET` requests, even if you add the `[BindProperty]` attribute. For security reasons, only requests using verbs like `POST` and `PUT` are bound. If you *do* want to bind `GET` requests, then you can set the `SupportsGet` property on the `[BindProperty]` attribute to opt-in to model binding.

TIP To bind `PageModel` properties for `GET` requests, use the `SupportsGet` property of the attribute, for example `[BindProperty(SupportsGet = true)]`.

Which part is the binding model

Listing 6.1 shows a Razor Page that uses multiple "binding models": the `Category` property, the `Username` property, and the `ProductModel` property (in the `OnPost` handler) are all model bound.

Using multiple models in this way is fine, but I prefer to use an approach that keeps all the model binding in a single, nested, class, which I often call `InputModel`. With this approach, the Razor Page in listing 6.1 could be written as:

```
public class IndexModel: PageModel
{
    [BindProperty]
    public InputModel Input { get; set; }
    public void OnGet()
    {
    }

    public class InputModel
    {
        public string Category { get; set; }
        public string Username { get; set; }
    }
}
```

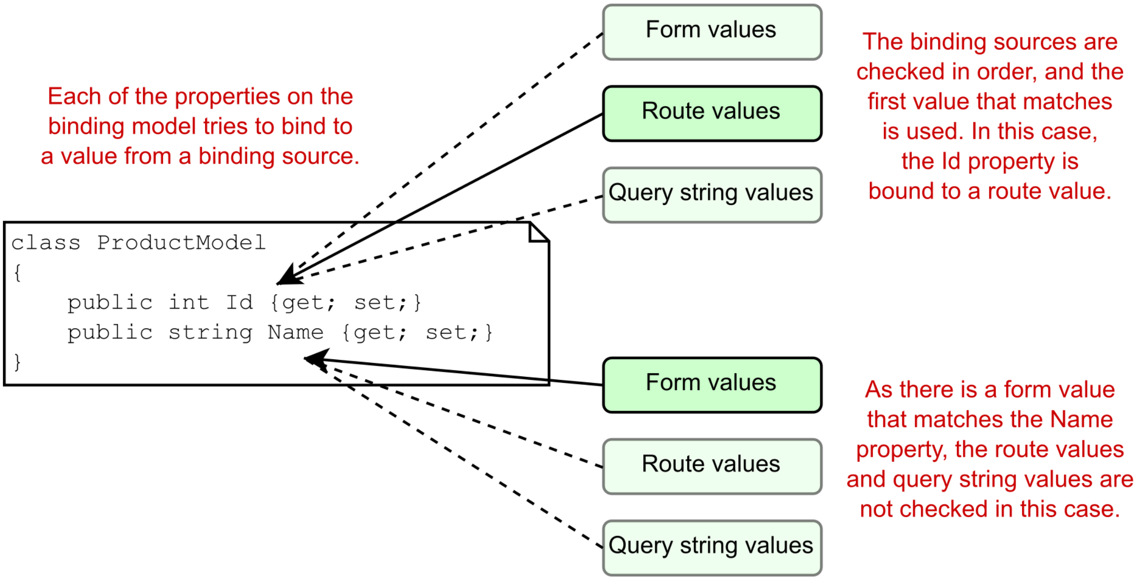
This approach has some organizational benefits that you'll learn more about in section 6.4.

ASP.NET Core automatically populates your binding models for you using properties of the request, such as the request URL, any headers sent in the HTTP request, any data explicitly POSTed in the request body, and so on.

By default, ASP.NET Core uses three different *binding sources* when creating your binding models. It looks through each of these in order and takes the first value it finds (if any) that matches the name of the binding model:

- *Form values*—Sent in the body of an HTTP request when a form is sent to the server using a POST.
- *Route values*—Obtained from URL segments or through default values after matching a route, as you saw in chapter 5.
- *Query string values*—Passed at the end of the URL, not used during routing.

The model binding process is shown in figure 6.3. The model binder checks each binding source to see if it contains a value that could be set on the model. Alternatively, the model can also choose the specific source the value should come from, as you'll see in section 6.2.3. Once each property is bound, the model is validated and is set as a property on the `PageModel` or is passed as a parameter to the page handler. You'll learn about the validation process in the second half of this chapter.



Each of the properties on the binding model tries to bind to a value from a binding source.

The binding sources are checked in order, and the first value that matches is used. In this case, the Id property is bound to a route value.

As there is a form value that matches the Name property, the route values and query string values are not checked in this case.

Figure 6.3 Model binding involves mapping values from binding sources, which correspond to different parts of a request.

`PageModel` properties or page handler parameters?

There are two different ways to use model binding in Razor Pages:

- Decorate properties on your `PageModel` with the `[BindProperty]` attribute.
- Add parameters to your page handler method.

Which of these approaches should you choose?

This answer to this question is largely a matter of taste. Setting properties on the `PageModel` and marking them with `[BindProperty]` is the approach you'll see most often in examples. If you use this approach, you'll be able to access the binding model when the view is rendered, as you'll see in chapters 7 and 8.

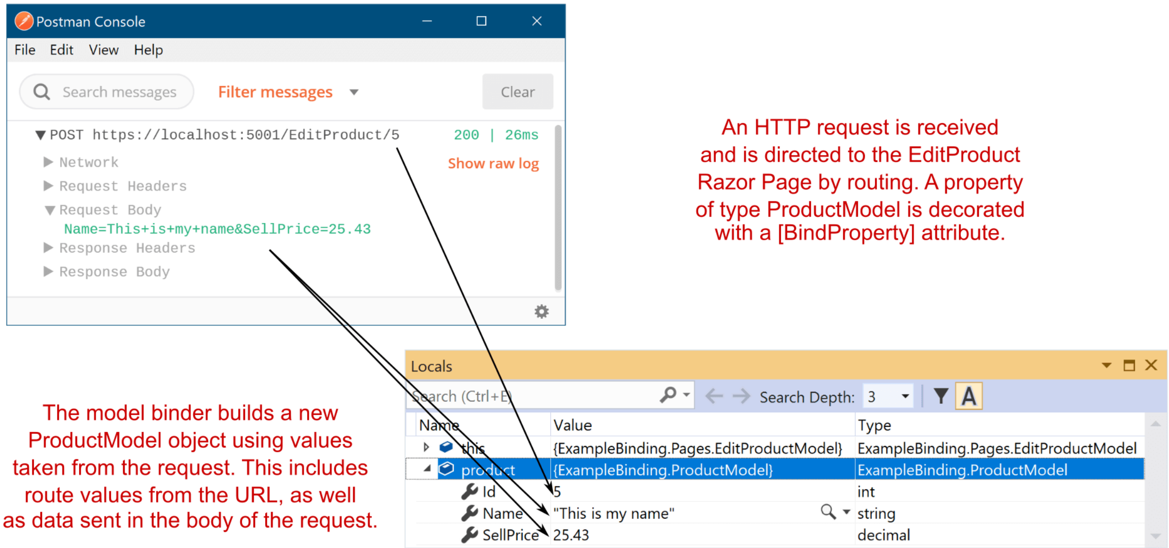
The alternative approach, adding parameters to page handler methods, provides more separation between the different MVC stages, because you won't be able to access the parameters outside of the page handler. On the downside, if you *do* need to display those values in the Razor view, you'll have to manually copy the parameters across to properties that *can* be accessed in the view.

The approach I choose tends to depend on the specific Razor Page I'm building. If I'm creating a form, I will favor the `[BindProperty]` approach, as I typically need access to the request values inside the Razor view. For simple pages, where the binding model is a product ID for example, I tend to favor the page handler parameter approach for its simplicity, especially if the handler is for a `GET` request. I give some more specific advice on my approach in section 6.4.

Figure 6.4 shows an example of a request creating the `ProductModel` method argument using model binding for the example shown at the start of this section:

```
public void OnPost(ProductModel product)
```

The `Id` property has been bound from a URL route parameter, but the `Name` and `SellPrice` properties have been bound from the request body. The big advantage of using model binding is that you don't have to write the code to parse requests and map the data yourself. This sort of code is typically repetitive and error-prone, so using the built-in conventional approach lets you focus on the important aspects of your application: the business requirements.



```
▼POST https://localhost:5001/EditProduct/5        200 | 26ms
  ▶ Network
  ▶ Request Headers
  ▼ Request Body
    Name=This+is+my+name&SellPrice=25.43
  ▶ Response Headers
  ▶ Response Body
```

Name	Value	Type
this	{ExampleBinding.Pages.EditProductModel}	ExampleBinding.Pages.EditProductModel
product	{ExampleBinding.ProductModel}	ExampleBinding.ProductModel
Id	5	int
Name	"This is my name"	string
SellPrice	25.43	decimal

An HTTP request is received and is directed to the EditProduct Razor Page by routing. A property of type ProductModel is decorated with a [BindProperty] attribute.

The model binder builds a new ProductModel object using values taken from the request. This includes route values from the URL, as well as data sent in the body of the request.

Figure 6.4 Using model binding to create an instance of a model that's used to execute a Razor Page.

TIP **Model binding is great for reducing repetitive code. Take advantage of it whenever possible and you'll rarely find yourself having to access the** `Request` **object directly.**

If you need to, the capabilities are there to let you completely customize the way model binding works, but it's relatively rare that you'll find yourself needing to dig too deep into this. For the majority of cases it works as-is, as you'll see in the remainder of this section.

6.2.1 Binding simple types

You'll start your journey into model binding by considering a simple Razor Page handler. The next listing shows a simple Razor Page that takes one number as a method parameter and squares it by multiplying the number by itself.

Listing 6.2 A Razor Page accepting a simple parameter

```
public class CalculateSquareModel : PageModel
{
    public void OnGet(int number)                  #A
    {
        Square = number * number;                  #B
    }

    public int Square { get; set; }                #C

}
```

#A The method parameter is the binding model.
#B A more complex example would do this work in an external service, in the application model.
#C The result is exposed as a property, and is used by the view to generate a response.

In the last chapter, you learned about routing, and how it selects a Razor Page to execute. You can update the route template for the Razor Page to be "CalculateSquare/{number}" by adding a {number} segment to the Razor Page's @page directive in the cshtml file, as we discussed in chapter 5:

```
@page "{number}"
```

When a client requests the URL /CalculatorSquare/5, the Razor Page framework uses routing to parse it for route parameters. This produces the route value pair:

```
number=5
```

The Razor Page's OnGet page handler contains a single parameter—an integer called number—which is your binding model. When ASP.NET Core executes this page handler method, it will spot the expected parameter, flick through the route values associated with the request, and find the number=5 pair. It can then bind the number parameter to this route value and execute the method. The page handler method itself doesn't care about where this value came from; it goes along its merry way, calculating the square of the value, and setting it on the Square property.

The key thing to appreciate is that you didn't have to write any extra code to try to extract the number from the URL when the method executed. All you needed to do was create a method parameter (or public property) with the right name and let model binding do its magic.

Route values aren't the only values the model binder can use to create your binding models. As you saw previously, the framework will look through three default *binding sources* to find a match for your binding models:

- Form values
- Route values
- Query string values

Each of these binding sources store values as name-value pairs. If none of the binding sources contain the required value, then the binding model is set to a new, default, instance of the type instead. The exact value the binding model will have in this case depends on the type of the variable:

- For value types, the value will be `default(T)`. For an `int` parameter this would be `0`, and for a `bool` it would be `false`.
- For reference types, the type is created using the default constructor. For custom types like `ProductModel`, that will create a new object. For nullable types like `int?` or `bool?`, the value will be `null`.
- For string types, the value will be `null`.

> **WARNING** **It's important to consider the behavior of your page handler when model binding fails to bind your method parameters. If none of the binding sources contain the value, the value passed to the method could be `null`, or have an unexpected default value.**

Listing 6.2 showed how to bind a *single* method parameter. Let's take the next logical step and look at how you'd bind *multiple* method parameters.

In the previous chapter, we discussed routing for building a currency converter application. As the next step in your development, your boss asks you to create a method in which the user provides a value in one currency and you must convert it to another. You first create a Razor Page called Convert.cshtml, and customize the route template for the page using the `@page` directive to use an absolute path containing two route values:

```
@page "/{currencyIn}/{ currencyOut}"
```

You then create a page handler that accepts the three values you need, as shown in the listing below.

Listing 6.3 A Razor Page handler accepting multiple binding parameters

```
public class ConvertModel : PageModel
{
    public void OnGet(
        string currencyIn,
        string currencyOut,
        int qty
)
    {
        /* method implementation */
    }
}
```

As you can see, there are three different parameters to bind. The question is, where will the values come from and what will they be set to? The answer is, it depends! Table 6.1 shows a whole variety of possibilities. All these examples use the same route template and page handler, but depending on the data sent, different values will be bound. The actual values might differ from what you expect, as the available binding sources offer conflicting values!

Table 6.1 Binding request data to page handler parameters from multiple binding sources

URL (route values)	HTTP body data (form values)	Parameter values bound

`/GBP/USD`		`currencyIn=GBP`
		`currencyOut=USD qty=0`

`/GBP/USD?currencyIn=CAD`	`QTY=50`	`currencyIn=GBP`
		`currencyOut=USD qty=50`

`/GBP/USD?qty=100`	`qty=50`	`currencyIn=GBP`
		`currencyOut=USD qty=50`

`/GBP/USD?qty=100`	`currencyIn=CAD&`
	`currencyOut=EUR& qty=50`
		`currencyIn=CAD`
		`currencyOut=EUR qty=50`

For each example, be sure you understand *why* the bound values have the values that they do. In the first example, the `qty` value isn't found in the form data, in the route values, or in the query string, so it has the default value of `0`. In each of the other examples, the request contains one or more duplicated values; in these cases, it's important to bear in mind the order in which the model binder consults the binding sources. By default, form values will take precedence over other binding sources, including route values!

> **NOTE** **The default model binder isn't case sensitive, so a binding value of** `QTY=50` **will happily bind to the** `qty` **parameter.**

Although this may seem a little overwhelming, it's relatively unusual to be binding from all these different sources at once. It's more common to have your values all come from the request body as form values, maybe with an ID from URL route values. This scenario serves as more of a cautionary tale about the knots you can twist yourself into if you're not sure how things work under the hood!

In these examples, you happily bound the `qty` integer property to incoming values, but as I mentioned earlier, the values stored in binding sources are all strings. What types can you convert a string to? The model binder will convert pretty much any primitive .NET type such as `int`, `float`, `decimal` (and `string` obviously), plus anything that has a `TypeConverter`.[23] There are a few other special cases that can be converted from a string, such as `Type`, but thinking of it as primitives only will get you a long way there!

[23] TypeConverters can be found in the System.ComponentModel.TypeConverter package. You can read more about them here: https://docs.microsoft.com/en-us/dotnet/standard/base-types/type-conversion#the-typeconverter-class.

6.2.2 Binding complex types

If it seems like only being able to bind simple primitive types is a bit limiting, then you're right! Luckily, that's not the case for the model binder. Although it can only convert strings *directly* to those primitive types, it's also able to bind complex types by traversing any properties your binding models expose.

In case this doesn't make you happy straight off the bat, let's look at how you'd have to build your page handlers if simple types were your only option. Imagine a user of your currency converter application has reached a checkout page and is going to exchange some currency. Great! All you need now is to collect their name, email, and phone number. Unfortunately, your page handler method would have to look something like this:

```
public IActionResult OnPost(string firstName, string lastName, string phoneNumber, string
       email)
```

Yuck! Four parameters might not seem that bad right now, but what happens when the requirements change and you need to collect other details? The method signature will keep growing. The model binder will bind the values quite happily, but it's not exactly clean code. Using the [BindProperty] approach doesn't really help either—we still have to clutter up our PageModel with lots of properties and attributes!

SIMPLIFYING METHOD PARAMETERS BY BINDING TO COMPLEX OBJECTS

A common pattern for any C# code when you have many method parameters is to extract a class that encapsulates the data the method requires. If extra parameters need to be added, you can add a new property to this class. This class becomes your binding model and might look something like this.

Listing 6.4 A binding model for capturing a user's details

```
public UserBindingModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string PhoneNumber { get; set; }
}
```

With this model, you can now update your page handler's method signature to

```
public IActionResult OnPost(UserBindingModel user)
```

Or alternatively, using the [BindProperty] approach, create a property on the PageModel:

```
[BindProperty]
public UserBindingModel User { get; set; }
```

And simplify the page handler signature even further:

```
public IActionResult OnPost()
```

Functionally, the model binder treats this new complex type a little differently. Rather than looking for parameters with a value that matches the parameter name (`user`, or `User` for the property), the model binder create a new instance of the model using `new UserBindingModel()`.

NOTE **You don't have to use custom classes for your methods; it depends on your requirements. If your page handler needs only a single integer, then it makes more sense to bind to the simple parameter.**

Next, the model binder loops through all the properties your binding model has, such as `FirstName` and `LastName` in listing 6.4, For each of these properties, it consults the collection of binding sources and attempts to find a name-value pair that matches. If it finds one, it sets the value on the property and moves on to the next.

TIP **Although the name of the model isn't necessary in this example, the model binder will also look for properties prefixed with the name of the property, such as** `user.FirstName` **and** `user.LastName` **for a property called** `User`**. You can use this approach when you have multiple complex parameters to a page handler, or multiple complex** `[BindProperty]` **properties. In general, for simplicity, you should avoid this situation if possible.**

Once all the properties that can be bound on the binding model are set, the model is passed to the page handler (or the `[BindProperty]` property is set), and the handler is executed as usual. The behavior from this point is identical to when you have lots of individual parameters—you'll end up with the same values set on your binding model—but the code is cleaner and easier to work with.

TIP **For a class to be model-bound, it must have a default public constructor. You can only bind properties which are public and settable.**

With this technique you can bind complex hierarchical models, whose properties are *themselves* complex models. As long as each property exposes a type that can be model-bound, the binder can traverse it with ease.

BINDING COLLECTIONS AND DICTIONARIES

As well as ordinary custom classes and primitives, you can bind to collections, lists, and dictionaries. Imagine you had a page in which a user selected all the currencies they were interested in; you'd display the rates for all those selected, as shown in figure 6.5.

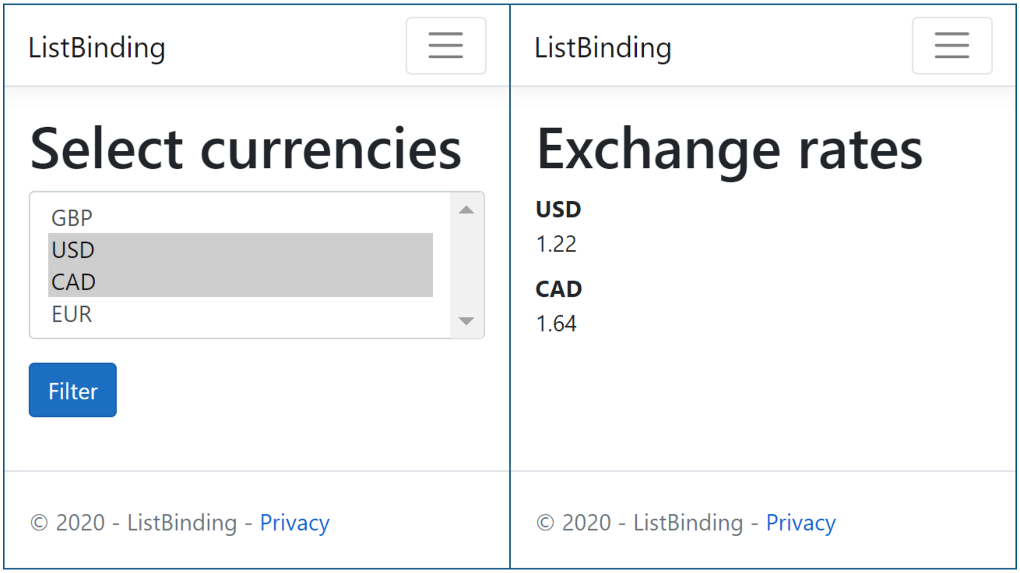


Figure 6.5 The select list in the currency converter application will send a list of selected currencies to the application. Model binding can bind the selected currencies and customize the view for the user to show the equivalent cost in the selected currencies.

To achieve this, you could create a page handler that accepts a `List<string>` type such as:

```
public void OnPost(List<string> currencies);
```

You could then `POST` data to this method by providing values in several different formats:

- `currencies[index]`—Where `currencies` is the name of the parameter to bind and `index` is the index of the item to bind, for example, `currencies[0]=GBR&currencies[1]=USD`.
- `[index]`—If you're only binding to a single list (as in this example), you can omit the name of the parameter, for example, `[0]=GBR&[1]=USD`.
- `currencies`—Alternatively, you can omit the `index` and send `currencies` as the key for every value, for example, `currencies=GBR&currencies=USD`.

The key values can come from route values and query values, but it's far more common to `POST` them in a form. Dictionaries can use similar binding, where the dictionary key replaces the index both when the parameter is named and when it's omitted.

If this all seems a bit confusing, don't feel too alarmed. If you're building a traditional web application, and using Razor views to generate HTML, then the framework will take care of generating the correct names for you. As you'll see in chapter 8, the Razor view will ensure that any form data you `POST` will be generated in the correct format.

BINDING FILE UPLOADS WITH IFORMFILE

A common feature of many websites is the ability to upload files. This could be a relatively infrequent activity, for example if a user uploads a profile picture for their Stack Overflow profile, or it may be integral to the application, like uploading photos to Facebook.

Letting users upload files to your application

Uploading files to websites is a pretty common activity, but you should carefully consider whether your application *needs* that ability. Whenever files can be uploaded by users the road is fraught with danger.

You should be careful to treat the incoming files as potentially malicious, don't trust the filename provided, take care of large files being uploaded, and don't allow the files to be executed on your server.

Files also raise questions as to where the data should be stored—should they go in a database, in the filesystem, or some other storage? None of these questions has a straightforward answer and you should think hard about the implications of choosing one over the other. Better yet, if you can avoid it, don't let users upload files!

ASP.NET Core supports uploading files by exposing the `IFormFile` interface. You can use this interface as your binding model, either as a method parameter to your page handler, or using the `[BindProperty]` approach, and it will be populated with the details of the file upload:

```
public void OnPost(IFormFile file);
```

You can also use an `IEnumerable<IFormFile>` if your need to accept multiple files:

```
public void OnPost(IEnumerable<IFormFile> file);
```

The `IFormFile` object exposes several properties and utility methods for reading the contents of the uploaded file, some of which are shown here:

```
public interface IFormFile
{
    string ContentType { get; }
    long Length { get; }
    string FileName { get; }
    Stream OpenReadStream();
}
```

As you can see, this interface exposes a `FileName` property, which returns the filename that the file was uploaded with. But you know not to trust users, right? You should *never* use the filename directly in your code—always generate a new filename for the file before you save it anywhere.

> **WARNING** Never use posted filenames in your code. Users can use them to attack your website and access files they shouldn't be able to.

The `IFormFile` approach is fine if users are only going to be uploading small files. When your method accepts an `IFormFile` instance, the whole content of the file is buffered in memory and on disk before you receive it. You can then use the `OpenReadStream` method to read the data out.

If users post large files to your website, you may find you start to run out of space in memory or on disk, as it buffers each of the files. In that case, you may need to stream the files directly to avoid saving all the data at once. Unfortunately, unlike the model binding approach, streaming large files can be complex and error-prone, so it's outside the scope of this book. For details, see the documentation at http://mng.bz/SH7X.

> **TIP** **Don't use the `IFormFile` interface to handle large file uploads as you may see performance issues. Be aware that you can't rely on users *not* to upload large files, so better yet, avoid file uploads entirely!**

For the vast majority of Razor Pages, the default configuration of model binding for simple and complex types works perfectly well, but you may find some situations where you need to take a bit more control. Luckily, that's perfectly possible, and you can completely override the process if necessary, by replacing the `ModelBinders` used in the guts of the framework.

However, it's rare to need that level of customization—I've found it's more common to want to specify which *binding source* to use for a page's binding model instead.

6.2.3 Choosing a binding source

As you've already seen, by default the ASP.NET Core model binder will attempt to bind your binding models from three different binding sources: form data, route data, and the query string.

Occasionally, you may find it necessary to specifically declare which binding source to bind to. In other cases, these three sources won't be sufficient at all. The most common scenarios are when you want to bind a method parameter to a request header value, or when the body of a request contains JSON-formatted data that you want to bind to a parameter. In these cases, you can decorate your binding models with attributes that say where to bind from, as shown in the listing below.

Listing 6.5 Choosing a binding source for model binding

```
public class PhotosModel: PageModel
{
    public void OnPost(
        [FromHeader] string userId,                  #A
        [FromBody] List<Photo> photos)               #B
    {
        /* method implementation */
    }
}
```

#A The userId will be bound from an HTTP header in the request.
#B The list of photos will be bound to the body of the request, typically in JSON format.

In this example, a page handler updates a collection of photos with a user ID. There are method parameters for the ID of the user to tag in the photos, `userId`, and a list of `Photo` objects to tag, `photos`.

Rather than binding these method parameters using the standard binding sources, I've added attributes to each parameter, indicating the binding source to use. The `[FromHeader]` attribute has been applied to the `userId` parameter. This tells the model binder to bind the value to an HTTP request header value called `userId`.

We're also binding a list of photos to the body of the HTTP request by using the `[FromBody]` attribute. This will read JSON from the body of the request and will bind it to the `List<Photo>` method parameter.

> **WARNING** Developers familiar with the previous version of ASP.NET should take note that the `[FromBody]` attribute is explicitly required when binding to JSON requests in Razor Pages. This differs from previous ASP.NET behavior, in which no attribute was required.

You aren't limited to binding JSON data from the request body—you can use other formats too, depending on which `InputFormatter`s you configure the framework to use. By default, only a JSON input formatter is configured. You'll see how to add an XML formatter in chapter 9, when I discuss Web APIs.

You can use a few different attributes to override the defaults and to specify a binding source for each binding model (or each property on the binding model):

- `[FromHeader]`—Bind to a header value
- `[FromQuery]`—Bind to a query string value
- `[FromRoute]`—Bind to route parameters
- `[FromForm]`—Bind to form data posted in the body of the request
- `[FromBody]`—Bind to the request's body content

You can apply each of these to any number of handler method parameters or properties, as you saw in listing 6.5, with the exception of the `[FromBody]` attribute—only one value may be decorated with the `[FromBody]` attribute. Also, as form data is sent in the body of a request, the `[FromBody]` and `[FromForm]` attributes are effectively mutually exclusive.

> **TIP** Only one parameter may use the `[FromBody]` attribute. This attribute will consume the incoming request as HTTP request bodies can only be safely read once.

As well as these attributes for specifying binding sources, there are a few other attributes for customizing the binding process even further:

- [BindNever]—The model binder will skip this parameter completely.[24]
- [BindRequired]—If the parameter was not provided, or was empty, the binder will add a validation error.
- [FromServices]—Used to indicate the parameter should be provided using dependency injection (see chapter 10 for details).

In addition, you have the [ModelBinder] attribute, which puts you into "God mode" with respect to model binding. With this attribute, you can specify the exact binding source, override the name of the parameter to bind to, and specify the type of binding to perform. It'll be rare that you need this one, but when you do, at least it's there!

By combining all these attributes, you should find you're able to configure the model binder to bind to pretty much any request data your page handler wants to use. In general, though, you'll probably find you rarely need to use them; the defaults should work well for you in most cases.

That brings us to the end of this section on model binding. If all has gone well, your page handler should have access to a populated binding model, and it's ready to execute its logic. It's time to handle the request, right? Nothing to worry about?

Not so fast! How do you know that the data you received was valid? That you haven't been sent malicious data attempting a SQL injection attack, or a phone number full of letters?

The binder is relatively blindly assigning values sent in a request, which you're happily going to plug into your own methods? What's to stop nefarious little Jimmy from sending malicious values to your application?

Except for basic safeguards, there's nothing stopping him, which is why it's important that you *always* validate the input coming in. ASP.NET Core provides a way to do this in a declarative manner out of the box, which is the focus of the second half of this chapter.

6.3 Handling user input with model validation

In this section I discuss:

- What is validation, and why do you need it?
- Using DataAnnotation attributes to describe the data you expect
- How to validate your binding models in page handlers

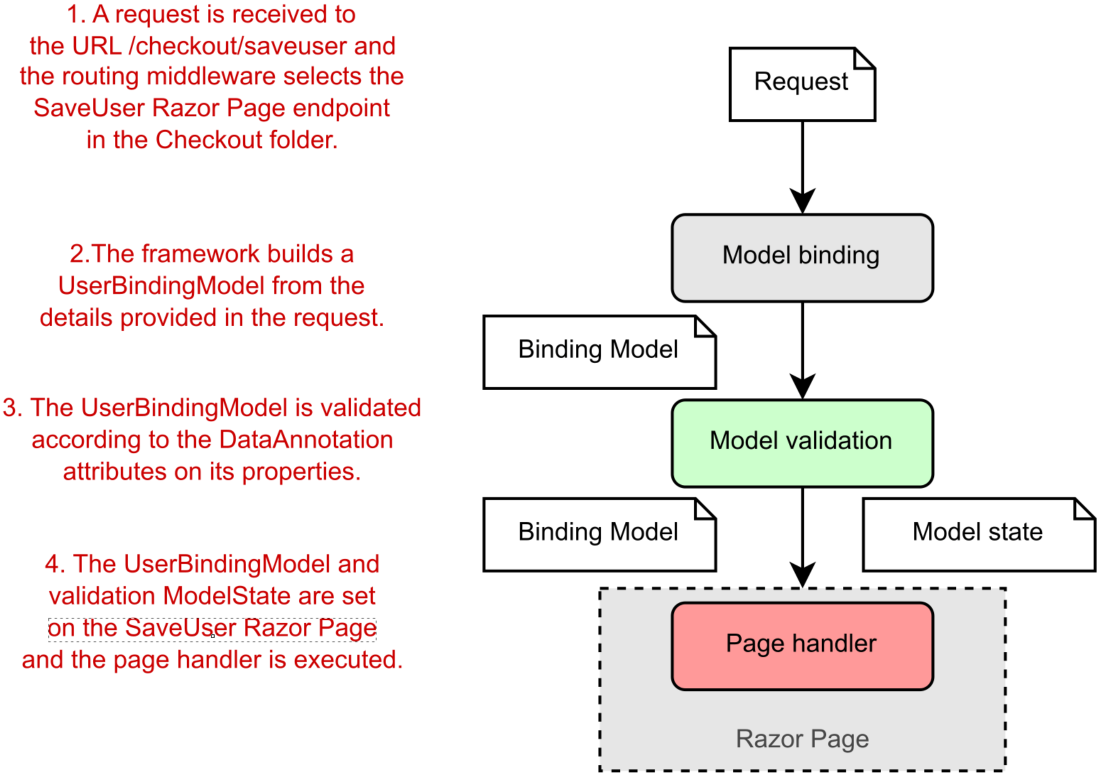
Validation in general is a pretty big topic, and one that you'll need to consider in every app you build. ASP.NET Core makes it relatively easy to add validation to your applications by making it an integral part of the framework.

[24] You can use the [BindNever] attribute to prevent mass assignment, as discussed in these two posts: http://mng.bz/QvfG and https://andrewlock.net/preventing-mass-assignment-or-over-posting-with-razor-pages-in-asp-net-core/.

6.3.1 The need for validation

Data can come from many different sources in your web application—you could load it from files, read it from a database, or you could accept values that a user typed into a form in requests. Although you might be inclined to trust that the data already on your server is valid (though this is sometimes a dangerous assumption!), you *definitely* shouldn't trust the data sent as part of a request.

Validation occurs in the Razor Pages framework after model binding, but before the page handler executes, as you saw in figure 6.2. Figure 6.6 shows a more compact view of where model validation fits in this process, demonstrating how a request to a checkout page that requests a user's personal details is bound and validated.



1. A request is received to the URL /checkout/saveuser and the routing middleware selects the SaveUser Razor Page endpoint in the Checkout folder.

2.The framework builds a UserBindingModel from the details provided in the request.

3. The UserBindingModel is validated according to the DataAnnotation attributes on its properties.

4. The UserBindingModel and validation ModelState are set on the SaveUser Razor Page and the page handler is executed.

Figure 6.6 Validation occurs after model binding but before the page handler executes. The page handler executes whether or not validation is successful.

You should always validate data provided by users before you use it in your methods. You have no idea what the browser may have sent you. The classic example of "little Bobby

Tables" (https://xkcd.com/327/) highlights the need to always validate any data sent by a user.

Validation isn't only to check for security threats, though; it's also needed to check for non-malicious errors, such as:

- Data should be formatted correctly (email fields have a valid email format).
- Numbers might need to be in a particular range (you can't buy -1 copies of this book!).
- Some values may be required but others are optional (name may be required for a profile but phone number is optional).
- Values must conform to your business requirements (you can't convert a currency to itself, it needs to be converted to a difference currency).

It might seem like some of these can be dealt with easily enough in the browser—for example, if a user is selecting a currency to convert to, don't let them pick the same currency; and we've all seen the "please enter a valid email address" messages.

Unfortunately, although this *client-side validation* is useful for users, as it gives them instant feedback, you can never rely on it, as it will *always* be possible to bypass these browser protections. It's always necessary to validate the data as it arrives at your web application, using *server-side validation*.

WARNING Always validate user input on the server-side of your application.

If that feels a little redundant, like you'll be duplicating logic and code, then I'm afraid you're right. It's one of the unfortunate aspects of web development; the duplication is a necessary evil. Thankfully, ASP.NET Core provides several features to try to reduce this burden.

TIP Blazor, the new C# SPA framework promises to solve some of these issues, but it's out of the scope of this book. For details, see https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor and *Blazor in Action* by Chris Sainty (Manning, 2021)

If you had to write this validation code fresh for every app, it would be tedious and likely error prone. Luckily, you can simplify your validation code significantly using a set of attributes provided by .NET Core.

6.3.2 Using DataAnnotations attributes for validation

Validation attributes, or more precisely `DataAnnotations` attributes, allow you to specify the rules that your binding model should conform to. They provide *metadata* about your model by describing the *sort* of data the binding model should contain, as opposed to the data itself.

DEFINITION *Metadata* describes other data, specifying the rules and characteristics the data should adhere to.

You can apply `DataAnnotations` attributes directly to your binding models to indicate the type of data that's acceptable. This allows you to, for example, check that required fields have been provided, that numbers are in the correct range, and that email fields are valid email addresses.

As an example, let's consider the checkout page for your currency converter application. You need to collect details about the user before you can continue, so you ask them to provide their name, email and, optionally, a phone number. The following listing shows the `UserBindingModel` decorated with validation attributes that represent the validation rules for the model. This expands on the example you saw in listing 6.4.

Listing 6.6 Adding `DataAnnotations` to a binding model to provide metadata

```
public class UserBindingModel
{
    [Required]                                          //#A
    [StringLength(100)]                                 //#B
    [Display(Name = "Your name")]                       //#C
    public string FirstName { get; set; }

    [Required]                                          //#A
    [StringLength(100)]                                 //#B
    [Display(Name = "Last name")]                       //#C
    public string LastName { get; set; }

    [Required]                                          //#A
    [EmailAddress]                                      //#D
    public string Email { get; set; }

    [Phone]
    [Display(Name = "Phone number")]                    //#C
    public string PhoneNumber { get; set; }
}
```

#A Values marked Required must be provided.
#B The StringLengthAttribute sets the maximum length for the property.
#C Customizes the name used to describe the property
#D Validates the value of Email is a valid email address

Suddenly your binding model contains a whole wealth of information where previously it was pretty sparse on details. For example, you've specified that the `FirstName` property should always be provided, that it should have a maximum length of 100 characters, and that when it's referred to (in error messages, for example) it should be called `"Your name"` instead of `"FirstName"`.

The great thing about these attributes is that they clearly declare the *expected* state of the model. By looking at these attributes, you know what the properties will/should contain. They also provide hooks for the ASP.NET Core framework to validate that the data set on the model during model binding is valid, as you'll see shortly.

You've got a plethora of attributes to choose from when applying `DataAnnotations` to your models. I've listed some of the common ones here, but you can find more in the

System.ComponentModel.DataAnnotations namespace. For a more complete list, I recommend using IntelliSense in Visual Studio/Visual Studio Code, or you can always look at the source code directly on GitHub (https://github.com/dotnet/runtime/tree/master/src/libraries/System.ComponentModel.Annotations).

- [CreditCard]—Validates that a property has a valid credit card format.
- [EmailAddress]—Validates that a property has a valid email address format.
- [StringLength(max)]—Validates that a string has at most max number of characters.
- [MinLength(min)]—Validates that a collection has at least the min number of items.
- [Phone]—Validates that a property has a valid phone number format.
- [Range(min, max)]—Validates that a property has a value between min and max.
- [RegularExpression(regex)]—Validates that a property conforms to the regex regular expression pattern.
- [Url]—Validates that a property has a valid URL format.
- [Required]—Indicates the property must not be null.
- [Compare]—Allows you to confirm that two properties have the same value (for example, Email and ConfirmEmail).

> **WARNING** **The [EmailAddress] and other attributes only validate that the *format* of the value is correct. They don't validate that the email address exists.**[25]

The DataAnnotations attributes aren't a new feature—they have been part of the .NET Framework since version 3.5—and their usage in ASP.NET Core is almost the same as in the previous version of ASP.NET.

They're also used for other purposes, in addition to validation. Entity Framework Core (among others) uses DataAnnotations to define the types of columns and rules to use when creating database tables from C# classes. You can read more about Entity Framework Core in chapter 12, and in Jon P Smith's *Entity Framework Core in Action, Second Edition* (Manning, 2021).

If the DataAnnotation attributes provided out of the box don't cover everything you need, then it's also possible to write custom attributes by deriving from the base ValidationAttribute. You'll see how to create a custom attribute for your currency converter application in chapter 19.

Alternatively, if you're not a fan of the attribute-based approach, ASP.NET Core is flexible enough that you can completely replace the validation infrastructure with your preferred technique. For example, you could use the popular FluentValidation library

[25] The phone number attribute is particularly lenient in the formats it allows. For an example of this, and how to do more rigorous phone number validation, see this post https://www.twilio.com/blog/validating-phone-numbers-effectively-with-c-and-the-net-frameworks.

(https://github.com/JeremySkinner/FluentValidation) in place of the DataAnnotations attributes if you prefer.

TIP DataAnnotations **are good for input validation of properties in isolation, but not so good for validating business rules. You'll most likely need to perform this validation outside the** DataAnnotations **framework.**

Whichever validation approach you use, it's important to remember that these techniques don't protect your application by themselves. The Razor Pages framework will ensure that validation occurs, but it doesn't automatically do anything if validation fails. In the next section, we look at how to check the validation result on the server and handle the case where validation has failed.

6.3.3 Validating on the server for safety

Validation of the binding model occurs before the page handler executes, but note that the handler *always* executes, whether the validation failed or succeeded. It's the responsibility of the page handler to check the result of the validation.

NOTE Validation happens automatically but handling validation failures is the responsibility of the page handler.

The Razor Pages framework stores the output of the validation attempt in a property on the PageModel called ModelState. This property is a ModelStateDictionary object, which contains a list of all the validation errors that occurred after model binding, as well as some utility properties for working with it.

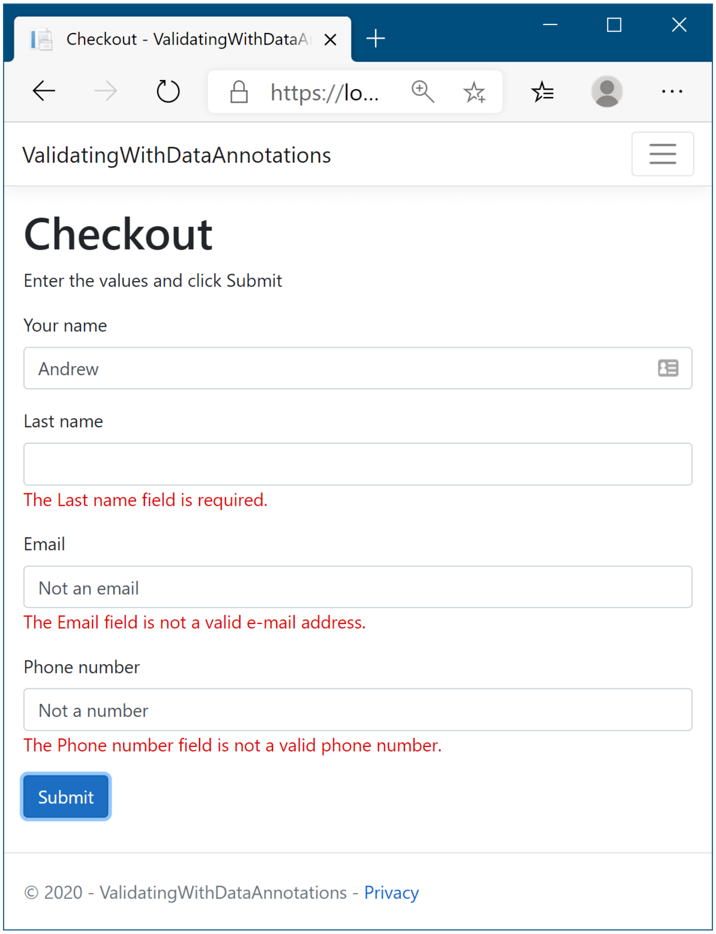
As an example, the listing below shows the OnPost page handler for the Checkout.cshtml Razor Page. The Input property is marked for binding, and uses the UserBindingModel type shown previously in listing 6.6. This page handler doesn't do anything with the data currently, but the pattern of checking ModelState early in the method is the key takeaway here.

Listing 6.7 Checking model state to view the validation result

```
public class CheckoutModel : PageModel                           #A
{
    [BindProperty]                                               #B
    public UserBindingModel Input { get; set; }                  #B

    public IActionResult OnPost()                                 #C
    {
        if (!ModelState.IsValid)                                  #D
        {
            return Page();                                        #E
        }

        /* Save to the database, update user, return success */    #F

        return RedirectToPage("Success");
    }
}
```

#A The ModelState property is available on the PageModel base class.
#B The Input property contains the model-bound data.
#C The binding model is validated before the page handler is executed.
#D If there were validation errors, IsValid will be false.
#E Validation failed, so redisplay the form with errors, and finish the method early.
#F Validation passed, so it’s safe to use the data provided in model.

If the `ModelState` indicates an error occurred, the method immediately calls the `Page` helper method. This returns a `PageResult` that will ultimately generate HTML to return to the user, as you saw in chapter 4. The view uses the (invalid) values provided in the `Input` property to repopulate the form when it’s displayed, as shown in figure 6.7. Also, helpful messages for the user are added automatically using the validation errors in the `ModelState`.



Checkout

Enter the values and click Submit

Your name

Andrew

Last name

The Last name field is required.

Email

Not an email

The Email field is not a valid e-mail address.

Phone number

Not a number

The Phone number field is not a valid phone number.

Submit

© 2020 - ValidatingWithDataAnnotations - Privacy

Figure 6.7 When validation fails, you can redisplay the form to display `ModelState` validation errors to the user. Note the “Your name” field has no associated validation errors, unlike the other fields.

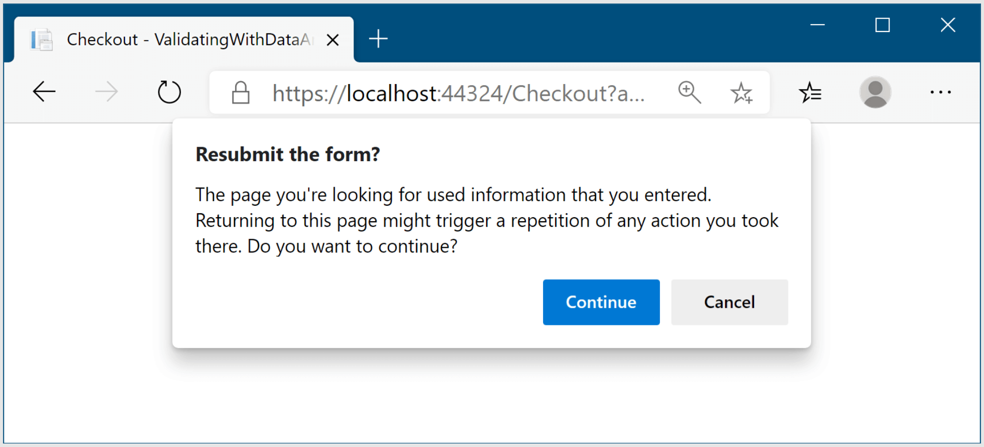
If the request is successful, the page handler returns a `RedirectToPage` result that redirects the user to the Success.cshtml Razor Page. This pattern of returning a redirect response after a successful POST is called the POST-REDIRECT-GET pattern.

> **NOTE** The error messages displayed on the form are the default values for each validation attribute. You can customize the message by setting the `ErrorMessage` property on any of the validation attributes. For example, you could customize a `[Required]` attribute using `[Required(ErrorMessage= "Required")]`.

POST-REDIRECT-GET

The POST-REDIRECT-GET design pattern is a web development pattern that prevents users from accidently submitting the same form multiple times. Users typically submit a form using the standard browser `POST` mechanism, sending data to the server. This is the normal way by which you might take a payment, for example.

If a server takes the naive approach and responds with a `200 OK` response and some HTML to display, then the user will still be on the same URL. If the user then refreshes their browser, they will be making an *additional* `POST` to the server, potentially making *another* payment! Browsers have some mechanisms to avoid this, such as in the following figure, but the user experience isn't desirable.


Checkout - ValidatingWithDataA
https://localhost:44324/Checkout?a...

Resubmit the form?

The page you're looking for used information that you entered. Returning to this page might trigger a repetition of any action you took there. Do you want to continue?

Refreshing a browser window after a `POST` causes a warning message to be shown to the user.

The POST-REDIRECT-GET pattern says that in response to a successful `POST`, you should return a `REDIRECT` response to a new URL, which will be followed by the browser making a `GET` to the new URL. If the user refreshes their browser now, then they'll be refreshing the final `GET` call to the new URL. No additional `POST` is made, so no additional payments or side effects should occur.

This pattern is easy to achieve in ASP.NET Core MVC applications using the pattern shown in listing 6.7. By returning a `RedirectToPageResult` after a successful `POST`, your application will be safe if the user refreshes the page in their browser.

You might be wondering why validation isn't handled automatically—if validation has occurred, and you have the result, why does the page handler get executed at all? Isn't there a risk that you might forget to check the validation result?

This is true, and in some cases the best thing to do is to make the generation of the validation check and response automatic. In fact, this is exactly the approach we will use for Web APIs when we cover them in chapter 9.

For Razor Pages apps however, you typically still want to generate an HTML response, even when validation failed. This allows the user to see the problem, and potentially correct it. This is much harder to make automatic.

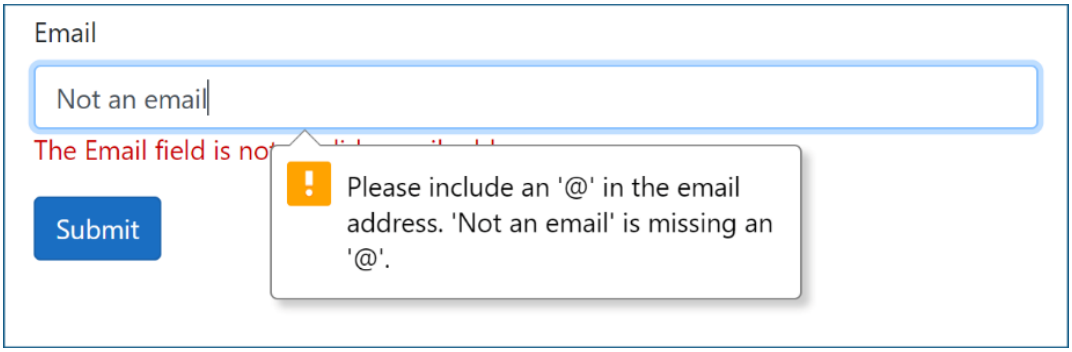
For example, you might find you need to load additional data before you can redisplay the Razor Page—such as loading a list of available currencies. That becomes simpler and more explicit with the `IsValid` pattern. Trying to do that automatically would likely end up with you fighting against edge-cases and workarounds.

Also, by including the `IsValid` check explicitly in your page handlers, it's easier to control what happens when *additional* validation checks fail. For example, if the user tries to update a product, then the `DataAnnotations` validation won't know whether a product with the requested ID exists, only whether the ID has the correct *format*. By moving the validation to the handler method, you can treat data and business rule validation failures in the same way.

I hope I've hammered home how important it is to validate user input in ASP.NET Core, but just in case: VALIDATE! There, we're good. Having said that, *only* performing validation on the server can leave users with a slightly poor experience. How many times have you filled out a form online, submitted it, gone to get a snack, and come back to find out you mistyped something and have to redo it. Wouldn't it be nicer to have that feedback immediately?

6.3.4 Validating on the client for user experience

You can add client-side validation to your application in a couple of different ways. HTML5 has several built-in validation behaviors that many browsers will use. If you display an email address field on a page and use the "email" HMTL input type, then the browser will automatically stop you from submitting an invalid format, as shown in figure 6.8.



Please include an '@' in the email address. 'Not an email' is missing an '@'.

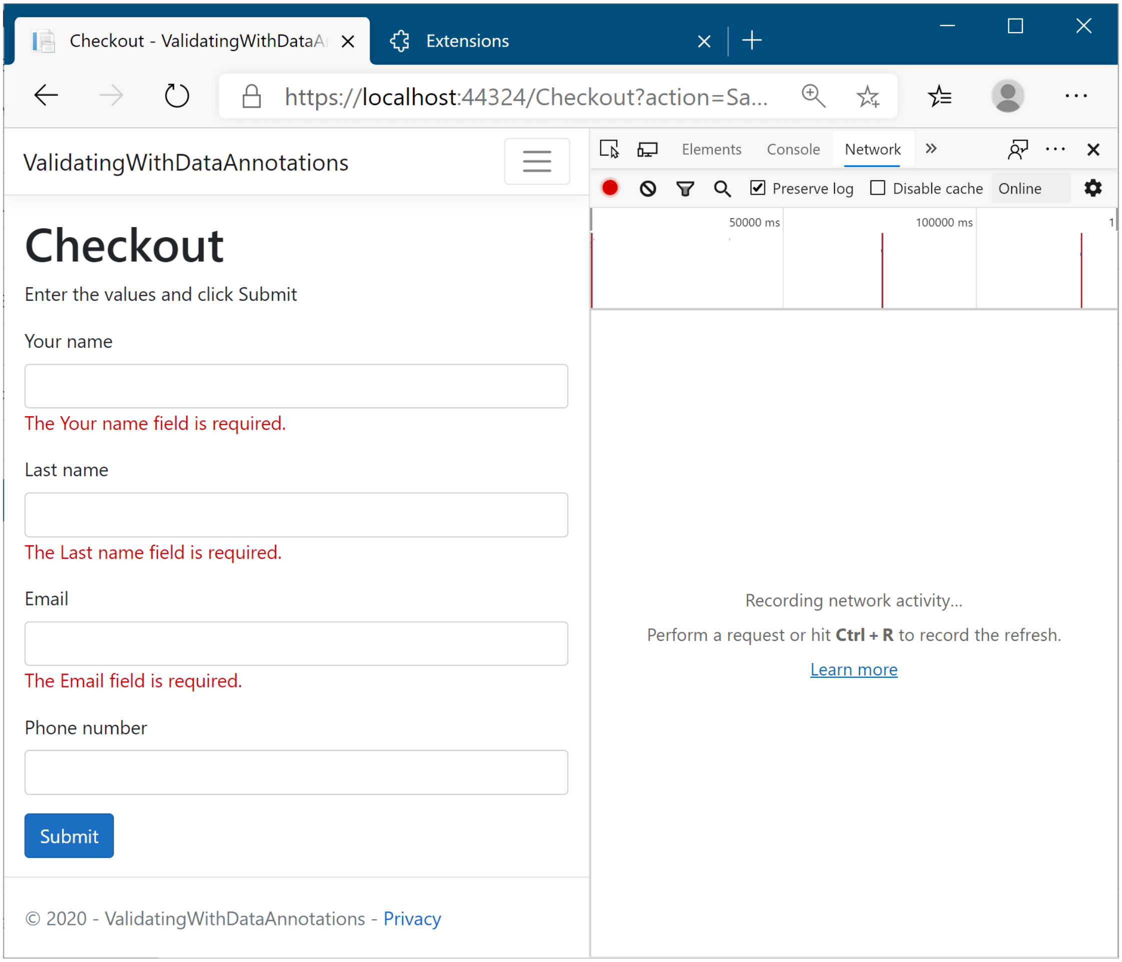
Figure 6.8 By default, modern browsers will automatically validate fields of the email type before a form is submitted.

Your application doesn't control this validation, it's built into modern HTML5 browsers.[26] The alternative approach is to perform client-side validation by running JavaScript on the page and checking the values the user has entered before submitting the form. This is the most common approach used in Razor Pages.

I'll go into detail on how to generate the client-side validation helpers in the next chapter, where you'll see the `DataAnnotations` attributes come to the fore once again. By decorating a view model with these attributes, you provide the necessary metadata to the Razor engine for it to generate the appropriate HTML.

With this approach, the user sees any errors with their form immediately, even before the request is sent to the server, as shown in figure 6.9. This gives a much shorter feedback cycle, providing a much better user experience.

[26] HTML5 constraint validation support varies by browser. For details on the available constraints, see http://mng.bz/daX3 and https://caniuse.com/#feat=constraint-validation.


Checkout - ValidatingWithDataA
Extensions
https://localhost:44324/Checkout?action=Sa...
ValidatingWithDataAnnotations
Elements
Console
Network
☑ Preserve log
□ Disable cache
Online
50000 ms
100000 ms
Submit
Learn more

Checkout

Enter the values and click Submit

Your name

The Your name field is required.

Last name

The Last name field is required.

Email

The Email field is required.

Phone number

© 2020 - ValidatingWithDataAnnotations - Privacy

Recording network activity...

Perform a request or hit **Ctrl + R** to record the refresh.

Figure 6.9 With client-side validation, clicking Submit will trigger validation to be shown in the browser before the request is sent to the server. As shown in the right-hand pane, no request is sent.

If you’re building an SPA, then the onus is on the client-side framework to validate the data on the client side before posting it to the Web API. The Web API will still validate the data when it arrives at the server, but the client-side framework is responsible for providing the smooth user experience.

When you use Razor Pages to generate your HTML, you get much of this validation for free. It automatically configures client-side validation for most of the built-in attributes without requiring additional work, as you’ll see in chapter 7. Unfortunately, if you’ve used custom `ValidationAttributes`, then these will only run on the server by default; you need to do some additional wiring up of the attribute to make it work on the client side too. Despite this, custom validation attributes can be useful for handling common validation scenarios in your application, as you'll see in chapter 19.

The model binding framework in ASP.NET Core gives you a lot of options on how to organize your Razor Pages: page handler parameters or `PageModel` properties; one binding model or multiple; where should you define your binding model classes? In the next section I give some advice on how *I* like to organize my Razor Pages.

6.4 Organizing your binding models in Razor Pages

In this section I give some general advice on how I like to configure the binding models in my Razor Pages. If you follow the patterns in this section, your Razor Pages will follow a consistent layout, making it easier for others to understand how each Razor Page in your app works.

> **NOTE** This advice is just personal preference, so feel free to adapt it if there are aspects you don't agree with. The important thing is to understand *why* I make each suggestion, and to take that on board. Where appropriate, I deviate from these guidelines too!

Model binding in ASP.NET Core has a lot of equivalent approaches to take, so there is no "correct" way to do it. The following listing shows an example of how I would design a simple Razor Page. This Razor Page displays a form for a product with a given ID and allows you to edit the details using a POST request. It's a much longer sample than we've looked at so far, but I highlight the important points below.

Listing 6.8 Designing an edit product Razor Page

```
public class EditProductModel : PageModel
{
    private readonly ProductService _productService;          #A
    public EditProductModel(ProductService productService)    #A
    {                                                         #A
        _productService = productService;                     #A
    }                                                         #A

    [BindProperty]                                            #B
    public InputModel Input { get; set; }                     #B

    public IActionResult OnGet(int id)                        #C
    {
        var product = _productService.GetProduct(id);         #D

        Input = new InputModel                                #E
        {                                                     #E
            Name = product.ProductName,                       #E
            Price = product.SellPrice,                        #E
        };                                                    #E
        return Page();                                        #E
    }

    public IActionResult OnPost(int id)                       #C
    {
        if (!ModelState.IsValid)                                 #F
        {                                                        #F
            return Page();                                       #F
        }                                                        #F

        _productService.UpdateProduct(id, Input.Name, Input.Price);    #G

        return RedirectToPage("Index");                          #H
    }


    public class InputModel                                     #I
    {                                                           #I
        [Required]                                              #I
        public string Name { get; set; }                        #I
                                                                #I
        [Range(0, int.MaxValue)]                                #I
        public decimal Price { get; set; }                      #I
    }                                                           #I
}
```

#A The ProductService is injected using DI and provides access to the application model
#B A single property is marked with BindProperty
#C The id parameter is model bound from the route template for both OnGet and OnPage handlers
#D Load the product details from the application model
#E Build an instance of the InputModel for editing in the form from the existing product's details
#F If the request was not valid, redisplay the form without saving
#G Update the product in the application model using the ProductService
#H Redirect to a new page using the POST-REDIRECT-GET pattern
#I Define the InputModel as a nested class in the Razor Page

This page shows the `PageModel` for a typical "edit form". These are very common in many line-of-business applications, among others, and is a scenario that Razor Pages works very well for. You'll see how to create the HTML side of forms in chapter 8.

> **NOTE The purpose of this example is only to highlight the model binding approach. The code is overly simplistic from a *logic* point of view. For example, it doesn't check that the product with the provided ID exists, or include any error handling.**

This form shows several patterns related to model binding that I try to adhere to when building Razor Pages:

- *Only bind a single property with* `[BindProperty]`. I favour having a single property decorated with `[BindProperty]` for model binding in general. When more than one value needs to be bound, I create a separate class, `InputModel`, to hold the values, and decorate that single property with `[BindProperty]`. Decorating a single property like this makes it harder to forget to add the attribute and means all of your Razor Pages use the same pattern.
- *Define your binding model as a nested class*. I define the `InputModel` as a nested class inside my Razor Page. The binding model is normally highly specific to that single page,

so doing this keeps everything your working on together. Additionally, I normally use that exact class name, InputModel for all my pages. Again, this adds consistency to your Razor Pages.

Don't use [BindProperties]. In addition to the [BindProperty] attribute, there is a [BindProperties] attribute (note the different spelling) that can be applied to the Razor Page PageModel directly. This will cause *all* properties in your model to be model bound, which can leave you open to over-posting attacks if you're not careful. I suggest you don't use the [BindProperties] attribute and stick to binding a *single* property with [BindProperty] instead.

- *Accept route parameters in the page handler*. For simple route parameters, such as the id passed into the OnGet and OnPost handlers in listing 6.8, I add parameters to the page handler method itself. This avoids the clunky SupportsGet=true syntax for GET requests.
- *Always validate before using data*. I said it before, so I'll say it again. Validate user input!

That concludes this look at model-binding in Razor Pages. You saw how the ASP.NET Core framework uses model binding to simplify the process of extracting values from a request and turning them into normal .NET objects you can quickly work with. The most important aspect of this chapter is the focus on validation—this is a common concern for all web applications, and the use of DataAnnotations can make it easy to add validation to your models.

In the next chapter, we continue our journey through Razor Pages by looking at how to create views. In particular, you'll learn how to generate HTML in response to a request using the Razor templating engine.

6.5 Summary

- Razor Pages uses three distinct "models", each responsible for a different aspect of a request. The binding models encapsulates data sent as part of a request. The application model represents the state of the application. The PageModel is the backing class for the Razor Page, and exposes the data used by the Razor view to generate a response.
- Model binding extracts values from a request and uses them to create .NET objects the page handler can use when they execute.
- Any properties on the PageModel marked with the [BindProperty] attribute, and method parameters of the page handlers, will take part in model binding.
- Properties decorated with [BindProperty] are not bound for GET requests. To bind GET requests, you must use [BindProperty(SupportsGet = true)] instead.
- By default, there are three binding sources: POSTed form values, route values, and the query string. The binder will interrogate these in order when trying to bind your binding models.

- When binding values to models, the names of the parameters and properties aren't case sensitive.
- You can bind to simple types or to the properties of complex types.
- To bind complex types, they must have a default constructor and public, settable properties.
- Simple types must be convertible to strings to be bound automatically, for example numbers, dates, and Boolean values.
- Collections and dictionaries can be bound using the `[index]=value` and `[key]=value` syntax, respectively.
- You can customize the binding source for a binding model using `[From*]` attributes applied to the method, such as `[FromHeader]` and `[FromBody]`. These can be used to bind to nondefault binding sources, such as headers or JSON body content.
- In contrast to the previous version of ASP.NET, the `[FromBody]` attribute is required when binding JSON properties (previously it was not required).
- Validation is necessary to check for security threats. Check that data is formatted correctly, confirm that it conforms to expected values and that it meets your business rules.
- ASP.NET Core provides `DataAnnotations` attributes to allow you to declaratively define the expected values.
- Validation occurs automatically after model binding, but you must manually check the result of the validation and act accordingly in your page handler by interrogating the `ModelState` property.
- Client-side validation provides a better user experience than server-side validation alone, but you should always use server-side validation.
- Client-side validation uses JavaScript and attributes applied to your HTML elements to validate form values.

7

Rendering HTML using Razor views

This chapter covers

- **Creating Razor views to display HTML to a user**
- **Using C# and the Razor markup syntax to generate HTML dynamically**
- **Reusing common code with layouts and partial views**

It's easy to get confused between the terms involved in Razor Pages—`PageModel`, page handlers, Razor views—especially as some of the terms describe concrete features, and others describe patterns and concepts. We've touched on all these terms in detail in previous chapters, but it's important to get them straight int your mind:

- Razor Pages—Razor Pages generally refers to the page-based paradigm which combines routing, model binding, and HTML generation using Razor views.
- Razor Page—A single Razor Page represents a single page or "endpoint". It typically consists of two files: a .cshtml file containing the Razor view, and a .cshtml.cs file containing the page's `PageModel`.
- `PageModel`—The `PageModel` for a Razor Page is where most of the action happens. It's where you define the binding models for a page, which extracts data from the incoming request. It's also where you define the page's page handlers.
- Page handler—Each Razor Page typically handles a single *route*, but it can handle multiple HTTP verbs like `GET` and `POST`. Each page handler typically handles a single HTTP verb.
- Razor view—Razor views (also called Razor templates) are used to generate HTML. They are typically used in the final stage of a Razor Page, to generate the HTML response to send back to the user.

In the previous four chapters, I've covered a whole cross-section of Razor Pages, including the MVC design pattern, the Razor Page `PageModel`, page handlers, routing, and binding models. This chapter covers the last part of the MVC pattern—using a view to generate the HTML that's delivered to the user's browser.

In ASP.NET Core, views are normally created using the *Razor* markup syntax (sometimes described as a templating language), which uses a mixture of HTML and C# to generate the final HTML. This chapter covers some of the features of Razor and how to use it to build the view templates for your application. Generally speaking, users will have two sorts of interactions with your app: they read data that your app displays, and they send data or commands back to it. The Razor language contains a number of constructs that make it simple to build both types of applications.

When displaying data, you can use the Razor language to easily combine static HTML with values from your `PageModel`. Razor can use C# as a control mechanism, so adding conditional elements and loops is simple, something you couldn't achieve with HTML alone.

The normal approach to sending data to web applications is with HTML forms. Virtually every dynamic app you build will use forms; some applications will be pretty much nothing *but* forms! ASP.NET Core and the Razor templating language include a number of helpers that make generating HTML forms easy, called *Tag Helpers*.

> **NOTE** You'll get a brief glimpse of Tag Helpers in the next section, but I'll explore them in detail in the next chapter.

In this chapter, we'll be focusing primarily on displaying data and generating HTML using Razor, rather than creating forms. You'll see how to render values from your `PageModel` to the HTML, and how to use C# to control the generated output. Finally, you'll learn how to extract the common elements of your views into sub-views called *layouts* and *partial views*, and how to compose them to create the final HTML page.

7.1 Views: rendering the user interface

In this section, I provide a quick introduction to rendering HTML using Razor views. We'll recap the MVC design pattern used by Razor Pages, and where the view fits in. You'll then see an introduction of how Razor syntax allows you to mix C# and HTML to generate dynamic UIs.

As you know from earlier chapters on the MVC design pattern, it's the job of the Razor Page's page handler to choose what to return to the client. For example, if you're developing a to-do list application, imagine a request to view a particular to-do item, as shown in figure 7.1.

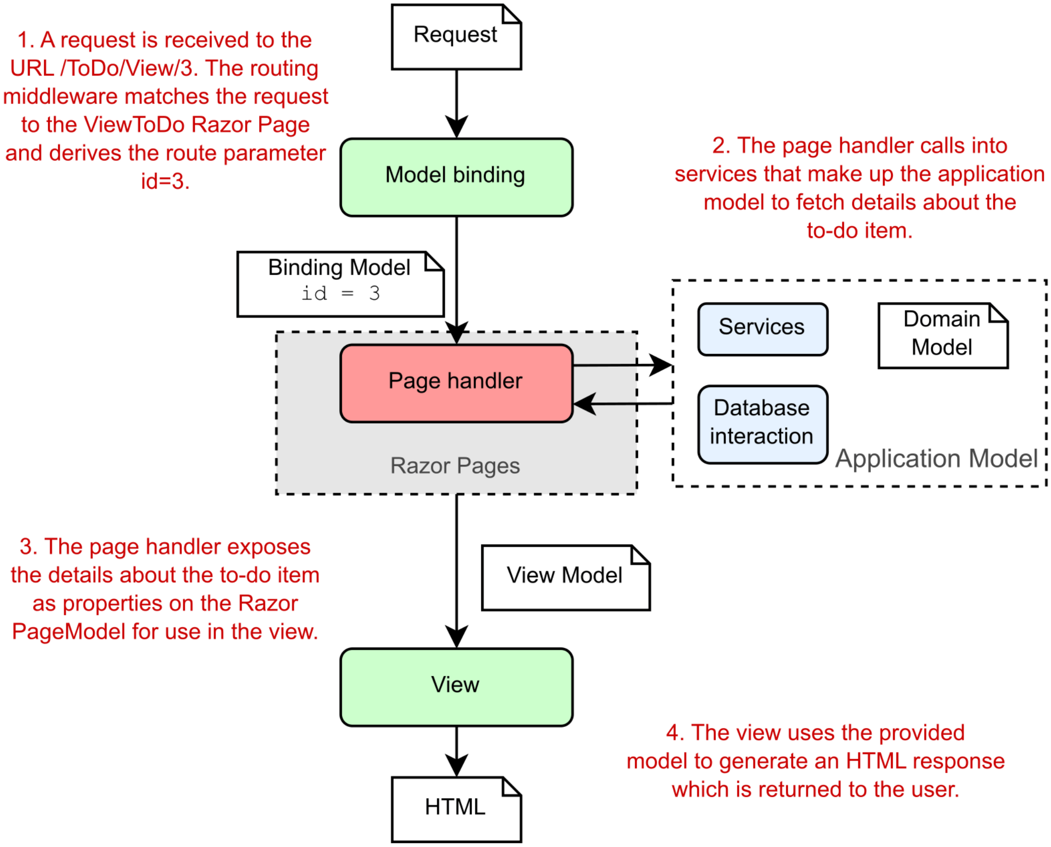

1. A request is received to the URL /ToDo/View/3. The routing middleware matches the request to the ViewToDo Razor Page and derives the route parameter id=3.
Request
Model binding
2. The page handler calls into services that make up the application model to fetch details about the to-do item.
Binding Model
id = 3
Services
Domain Model
Page handler
Database interaction
Razor Pages
Application Model
3. The page handler exposes the details about the to-do item as properties on the Razor PageModel for use in the view.
View Model
View
4. The view uses the provided model to generate an HTML response which is returned to the user.
HTML

Figure 7.1 Handling a request for a to-do list item using ASP.NET Core Razor Pages. The page handler builds the data required by the view and exposes it as properties on the PageModel. The view generates HTML based only on the data provided, it doesn't need to know where that data come from.

A typical request follows the steps shown in figure 7.1:

1. The middleware pipeline receives the request and the routing middleware determines the endpoint to invoke—in this case, the `ViewToDo` Razor Page.
2. The model binder (part of the Razor Pages framework) uses the request to build the binding models for the page, as you saw in the previous chapter. The binding models are set as properties on the Razor Page or are passed to the page handler method as arguments when the handler is executed. The page handler checks that you have passed a valid `id` for the to-do item, making the request valid.
3. Assuming all looks good, the page handler calls out to the various services that make up the application model. This might load the details about the to-do from a database, or from the filesystem, returning them to the handler. As part of this process, either

the application model or the page handler itself generates values to pass to the view and sets them as properties on the Razor Page PageModel.

Once the page handler has executed, the PageModel should contain all the data required to render a view. In this example, it contains details about the to-do itself, but it might also contain other data: how many to-dos you have left, whether you have any to-dos scheduled for today, your username, and so on, anything that controls how to generate the end UI for the request.

4. The Razor view template uses the PageModel to generate the final response and returns it back to the user via the middleware pipeline.

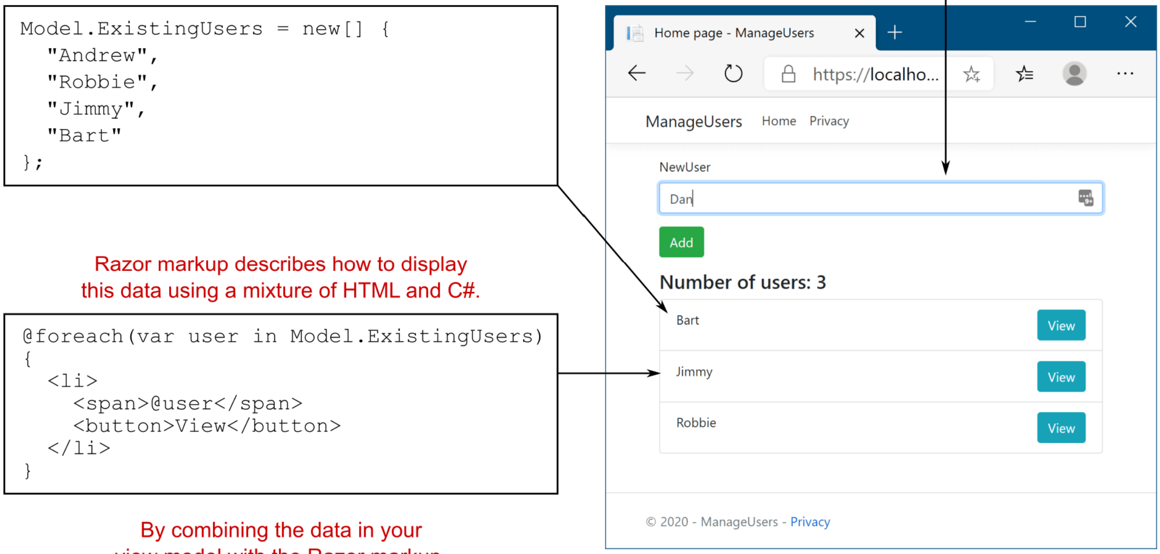
A common thread throughout this discussion of MVC is the separation of concerns MVC brings, and this is no different when it comes to your views. It would be easy enough to directly generate the HTML in your application model or in your controller actions, but instead you delegate that responsibility to a single component, the view.

But even more than that, you'll also separate the *data* required to build the view from the *process* of building it, by using properties on the PageModel. These properties should contain all the dynamic data needed by the view to generate the final output.

> **TIP** **Views shouldn't call methods on the PageModel—the view should generally only be accessing data that has already been collected and exposed as properties.**

Razor Page handlers indicate that the Razor view should be rendered by returning a PageResult (or by returning void), as you saw in chapter 4. The Razor Pages infrastructure executes the Razor view associated with a given Razor Page to generate the final response. The use of C# in the Razor template means you can dynamically generate the final HTML sent to the browser. This allows you to, for example, display the name of the current user in the page, hide links the current user doesn't have access to, or render a button for every item in a list.

Imagine your boss asks you to add a page to your application that displays a list of the application's users. You should also be able to view a user from the page, or create a new one, as shown in figure 7.2.



The PageModel contains the data you wish to display on the page.

```
Model.ExistingUsers = new[] {
  "Andrew",
  "Robbie",
  "Jimmy",
  "Bart"
};
```

Form elements can be used to send values back to the application.

Razor markup describes how to display this data using a mixture of HTML and C#.

```
@foreach(var user in Model.ExistingUsers)
{
  <li>
    <span>@user</span>
    <button>View</button>
  </li>
}
```

Home page - ManageUsers

https://localho...

NewUser

Dan

Number of users: 3

Bart

Jimmy

Robbie

© 2020 - ManageUsers - Privacy

By combining the data in your view model with the Razor markup, HTML can be generated dynamically, instead of being fixed at compile time.

Figure 7.2 The use of C# in Razor lets you easily generate dynamic HTML that varies at runtime. In this example, using a `foreach` loop inside the Razor view dramatically reduces the duplication in the HTML that you would otherwise have to write.

With Razor templates, generating this sort of dynamic content is simple. For example, the following listing shows the template that could be used to generate the interface in figure 7.2. It combines standard HTML with C# statements, and uses Tag Helpers to generate the form elements.

Listing 7.1 A Razor template to list users and a form for adding a new user

```
@page
@model IndexViewModel
<div class="row">                                           #A
<div class="col-md-6">                                      #A
<form method="post">
    <div class="form-group">
        <label asp-for="NewUser"></label>                   #B
        <input class="form-control" asp-for="NewUser" />    #B
        <span asp-validation-for="NewUser"></span>          #B
    </div>
    <div class="form-group">
        <button type="submit"
          class="btn btn-success">Add</button>
    </div>
</form>
</div>
</div>

<h4>Number of users: @Model.ExistingUsers.Count</h4>  #C
<div class="row">
<div class="col-md-6">
<ul class="list-group">
@foreach (var user in Model.ExistingUsers)            #D
{
<li class="list-group-item d-flex justify-content-between">
    <span>@user</span>
    <a class="btn btn-info"
        asp-page="ViewUser"                           #E
        asp-route-userName="@user">View</a>           #E
</li>
}
</ul>
</div>
</div>
```

#A Normal HTML is sent to the browser unchanged.
#B Tag Helpers attach to HTML elements to create forms.
#C Values can be written from C# objects to the HTML.
#D C# constructs like for loops can be used in Razor.
#E Tag Helpers can also be used outside of forms to help in other HTML generation.

This example demonstrates a variety of Razor features. There’s a mixture of HTML that’s written unmodified to the response output, and various C# constructs that are used to dynamically generate HTML. In addition, you can see several Tag Helpers. These look like normal HTML attributes that start `asp-`, but they’re part of the Razor language. They can customize the HTML element they’re attached to, changing how it’s rendered. They make building HTML forms much simpler than they would be otherwise. Don’t worry if this template is a bit overwhelming at the moment; we’ll break it all down as you progress through this chapter and the next.

Razor Pages are compiled when you build your application. Behind the scenes, they become just another C# class in your application. It’s also possible to enable *runtime* compilation of your Razor Pages. This allows you modify your Razor Pages while your app is running, without having to explicitly stop and rebuild. This can be handy when developing locally but is best avoided when you deploy to production[27].

> **NOTE** Like most things in ASP.NET Core, it’s possible to swap out the Razor templating engine and replace it with your own server-side rendering engine. You can’t replace Razor with a client-side framework like

[27] For details on enabling runtime compilation, including enabling conditional pre-compilation for production environments, see the documentation: https://docs.microsoft.com/en-us/aspnet/core/mvc/views/view-compilation.

AngularJS or React. If you want to take this approach, you'd use Web APIs instead. I'll discuss Web APIs in detail in chapter 9.

In the next section we'll look in more detail at how Razor views fit into the Razor Pages framework, and how to pass data from your Razor Page handlers to the Razor view to help build the HTML response.

7.2 Creating Razor views

In this section we look at how Razor views fit into the Razor Pages framework. You'll learn how to pass data from your page handlers to your Razor views, and how you can use that data to generate dynamic HTML.

With ASP.NET Core, whenever you need to display an HTML response to the user, you should use a view to generate it. Although it's possible to directly generate a `string` from your page handlers which will be rendered as HTML in the browser, this approach doesn't adhere to the MVC separation of concerns and will quickly leave you tearing your hair out.

NOTE Some middleware, such as the `WelcomePageMiddleware` you saw in chapter 3, may generate HTML responses without using a view, which can make sense in some situations. But your Razor Page and MVC controllers should always generate HTML using views.

Instead, by relying on Razor views to generate the response, you get access to a wide variety of features, as well as editor tooling to help! This section serves as a gentle introduction to Razor views, the things you can do with them, and the various ways you can pass data to them.

7.2.1 Razor views and code-behind

In this book you've already seen that Razor Pages typically consist of two files:

- The .cshtml file, commonly called the Razor view.
- The .cshtml.cs file, commonly called the "code-behind," which contains the `PageModel`.

The Razor view contains the `@page` directive which *makes it* a Razor Page, as you saw in chapter 4. Without this directive, the Razor Pages framework will not route requests to the page, and the file will be ignored for most purposes.

DEFINITION A *directive* is a statement in a Razor file that changes the way the template is parsed or compiled. Another common directive is the `@using newNamespace` directive, which would make objects in the `newNamespace` namespace available.

The code-behind .cshtml.cs file contains the `PageModel` for an associated Razor Page. It contains the page handlers that respond to requests, and it is where the Razor Page typically interacts with other parts of your application.

Even though the .cshtml and .cshtml.cs files share the same name, for example ToDoItem.cshtml and ToDoItem.cshtml.cs, it's not the filename that's linking them together. If it's not by filename, how does the Razor Pages framework know which PageModel is associated with a given Razor Page view file?

At the top of each Razor Page, just after the @page directive, is an @model directive with a Type, indicating which PageModel is associated with the Razor view. For example, the following directives indicate that the ToDoItemModel is the PageModel associated with the Razor Page:

```
@page
@model ToDoItemModel
```

Once a request is routed to a Razor Page, as we covered in chapter 5, the framework looks for the @model directive to decide which PageModel to use. Based on the PageModel selected, it then binds to any properties in the PageModel marked with the [BindProperty] attribute (as we covered in the chapter 6) and executes the appropriate page handler (based on the request's HTTP verb).

> **NOTE** Technically, the PageModel and @model directive are optional. If you don't specify a PageModel, the framework will execute a default page handler, as you saw in chapter 5. It's also possible to combine the .cshtml and .cshtml.cs files into a single .cshtml file. In practice, neither of these approaches are very common, even for simple pages, but it's something to be aware of if you run into it.[28]

In addition to the @page and @model directives, the Razor view file contains the Razor template that is executed to generate the HTML response.

7.2.2 Introducing Razor templates

Razor view templates contain a mixture of HTML and C# code interspersed with one another. The HTML markup lets you easily describe exactly what should be sent to the browser, whereas the C# code can be used to dynamically change what is rendered. For example, the listing below shows an example of Razor rendering a list of strings, representing to-do items.

Listing 7.2 Razor template for rendering a list of strings

```
@page
@{                                                  #A
    var tasks = new List<string>                    #A
      { "Buy milk", "Buy eggs", "Buy bread" };      #A
}                                                   #A
<h1>Tasks to complete</h1>                          #B
<ul>
```

[28] These alternative approaches are not generally considered idiomatic, so I don't discuss them in this book, but you can read more about them here: https://www.learnrazorpages.com/razor-pages.

```
@for(var i=0; i< tasks.Count; i++)                #C
{                                                 #C
  var task = tasks[i];                            #C
  <li>@i - @task</li>                             #C
}                                                 #C
</ul>
```

#A Arbitrary C# can be executed in a template. Variables remain in scope throughout the page.
#B Standard HTML markup will be rendered to the output unchanged.
#C Mixing C# and HTML allows you to dynamically create HTML at runtime.

The pure HTML sections in this template are the angle brackets. The Razor engine copies this HTML directly to the output, unchanged, as though you were writing a normal HTML file.

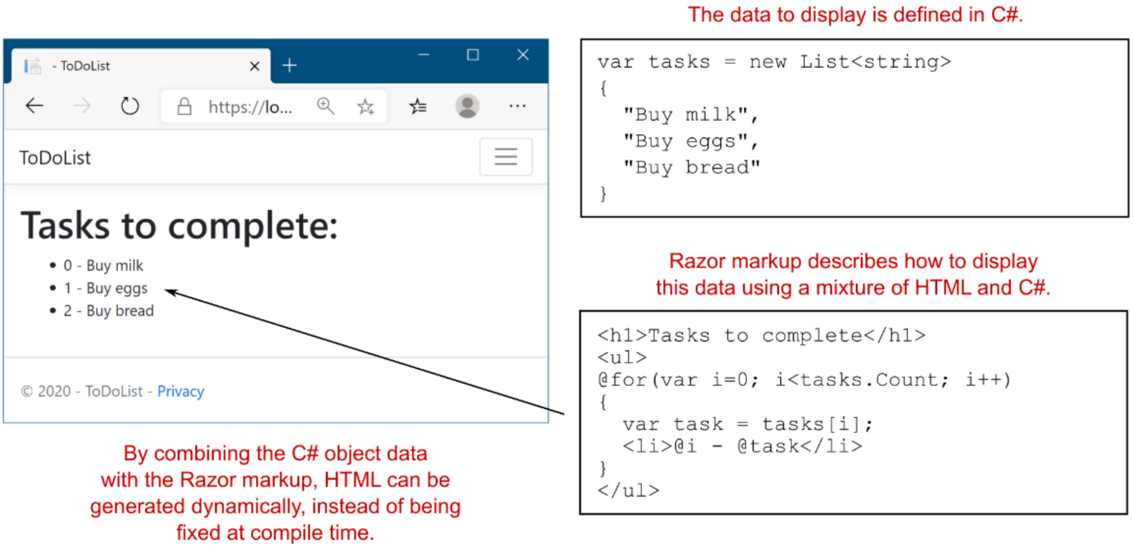
As well as HTML, you can also see a number of C# statements in there. The advantage of being able to, for example, use a `for` loop rather than having to explicitly write out each `<li>` element should be self-evident. I'll dive a little deeper into more of the C# features of Razor in the next section. When rendered, this template would produce the HTML shown here.

Listing 7.3 HTML output produced by rendering a Razor template

```
<h1>Tasks to complete</h1>        #A
<ul>                              #A
  <li>0 - Buy milk</li>           #B
  <li>1 - Buy eggs</li>           #B
  <li>2 - Buy bread</li>          #B
</ul>                             #C
```

#A HTML from the Razor template is written directly to the output.
#B The <li> elements are generated dynamically based on the data.
#C HTML from the Razor template is written directly to the output.

As you can see, the final output of a Razor template after it's been rendered is simple HTML. There's nothing complicated left, just straight HTML markup that can be sent to the browser and rendered. Figure 7.3 shows how a browser would render it.



The data to display is defined in C#.

```
var tasks = new List<string>
{
  "Buy milk",
  "Buy eggs",
  "Buy bread"
}
```

Razor markup describes how to display this data using a mixture of HTML and C#.

```
<h1>Tasks to complete</h1>
<ul>
@for(var i=0; i<tasks.Count; i++)
{
  var task = tasks[i];
  <li>@i - @task</li>
}
</ul>
```

Tasks to complete:

- 0 - Buy milk
- 1 - Buy eggs
- 2 - Buy bread

By combining the C# object data with the Razor markup, HTML can be generated dynamically, instead of being fixed at compile time.

Figure 7.3 Razor templates can be used to generate the HTML dynamically at runtime from C# objects. In this case, a `for` loop is used to create repetitive HTML `<li>` elements.

In this example, I hardcoded the list values for simplicity—there was no dynamic data provided. This is often the case on simple Razor Pages like you might have on your homepage—you need to display an almost static page. For the rest of your application, it will be far more common to have some sort of data you need to display, typically exposed as properties on your `PageModel`.

7.2.3 Passing data to views

In ASP.NET Core, you have several ways of passing data from a page handler in a Razor Page to its view. Which approach is best will depend on the data you're trying to pass through, but in general you should use the mechanisms in the following order:

- `PageModel` *properties*—You should generally expose any data that needs to be displayed as properties on your `PageModel`. Any data which is specific to the associated Razor view should be exposed this way. The `PageModel` object is available in the view when it's rendered, as you'll see shortly.
- `ViewData`—This is a dictionary of objects with `string` keys that can be used to pass arbitrary data from the page handler to the view. In addition, it allows you to pass data to *_layout* files, as you'll see in section 7.4. This is the main reason for using `ViewData` instead of setting properties on the `PageModel`.

- HttpContext—Technically the HttpContext object is available in both the page handler and Razor view, so you *could* use it to transfer data between them. But don't—there's no need for it with the other methods available to you.

Far and away the best approach for passing data from a page handler to a view is to use properties on the PageModel. There's nothing special about the properties themselves; you can store anything there to hold the data you require.

> **NOTE** **Many frameworks have the concept of a data context for binding UI components. The PageModel is a similar concept, in that it contains values to display in the UI, but the binding is only one-directional; the PageModel provides values to the UI, and once the UI is built and sent as a response, the PageModel is destroyed.**

As I described in section 7.2.1, the @model directive at the top of your Razor view describes which Type of PageModel is associated with a given Razor Page. The PageModel associated with a Razor Page contains one or more page handlers, and exposes data as properties for use in the Razor view.

Listing 7.4 Exposing data as properties on a PageModel

```
public class ToDoItemModel : PageModel                #A
{
    public List<string> Tasks { get; set; }           #B
    public string Title { get; set; }                 #B

    public void OnGet(int id)
    {
        Title = "Tasks for today",                 #C
        Tasks = new List<string>{                  #C
        {                                          #C
            "Get fuel",                            #C
            "Check oil",                           #C
            "Check tyre pressure"                  #C
        }                                          #C
    }
}
```

#A The PageModel is passed to the Razor view when it executes.
#B The public properties can be accessed from the Razor view.
#C Building the required data: this would normally call out to a service or database to load the data.

You can access the PageModel instance itself from the Razor view using the Model property. For example, to display the Title property of the ToDoItemModel in the Razor view, you'd use <h1>@Model.Title</h1>. This would render the string provided in the ToDoItemModel.Title property, producing the <h1>Tasks for today</h1> HTML.

> **TIP** **Note that the @model directive should be at the top of your view, just after the @page directive, and has a lowercase m. The Model property can be accessed anywhere in the view and has an uppercase M.**

In the vast majority of cases, using public properties on your `PageModel` is the way to go; it’s the standard mechanism for passing data between the page handler and the view. But in some circumstances, properties on your `PageModel` might not be the best fit. This is often the case when you want to pass data between view layouts (you’ll see how this works in section 7.4).

A common example is the title of the page. You need to provide a title for every page in your application, so you *could* create a base class with a `Title` property and make every `PageModel` inherit from it. But that’s very cumbersome, so a common approach for this situation is to use the `ViewData` collection to pass data around.

In fact, the standard Razor Page templates use this approach by default by setting values on the `ViewData` dictionary from within the view itself:

```
@{
    ViewData["Title"] = "Home Page";
}
<h2>@ViewData["Title"].</h2>
```

This template sets the value of the `"Title"` key in the `ViewData` dictionary to `"Home Page"` and then fetches the key to render in the template. This set and immediate fetch might seem superfluous, but as the `ViewData` dictionary is shared throughout the request, it makes the title of the page available in layouts, as you’ll see later. When rendered, this would produce the following output:

```
<h2>Home Page.</h2>
```

You can also set values in the `ViewData` dictionary from your page handlers in two different ways, as shown in the following listing.

Listing 7.5 Setting ViewData values using an attribute

```
public class IndexModel: PageModel
{
    [ViewData]                                    #A
    public string Title { get; set; }

    public void OnGet()
    {
        Title = "Home Page";                     #B
        ViewData["Subtitle"] = "Welcome";        #C
    }
}
```

#A Properties marked with the [ViewData] attribute are set in the ViewData.
#B The value of ViewData["Title"] will be set to "Home Page".
#C You can set keys in the ViewData dictionary directly.

You can display the values in the template in the same way as before:

```
<h1>@ViewData["Title"]</h3>
<h2>@ViewData["Subtitle"]</h3>
```

TIP **I don't find the `[ViewData]` attribute especially useful, but it's another feature to look out for. Instead, I create a set of global, static constants for any `ViewData` keys, and reference those, instead of typing `"Title"` repeatedly. You'll get IntelliSense for the values, they'll be refactor-safe, and you'll avoid hard-to-spot typos.**

As I mentioned previously, there are other mechanisms besides `PageModel` properties and `ViewData` that you can use to pass data around, but these two are the only ones I use personally, as you can do everything you need with them. As a reminder, always use `PageModel` properties where possible, as you benefit from strong typing and IntelliSense. Only fall back to `ViewData` for values that need to be accessed *outside* of your Razor view.

You've had a small taste of the power available to you in Razor templates, but in the next section, I'd like to dive a little deeper into some of the available C# capabilities.

7.3 Creating dynamic web pages with Razor

You might be glad to know that pretty much anything you can do in C# is possible in Razor syntax. Under the covers, the cshtml files are compiled into normal C# code (with `string` for the raw HTML sections), so whatever weird and wonderful behavior you need can be created!

Having said that, just because you *can* do something doesn't mean you *should*. You'll find it much easier to work with, and maintain, your files if you keep them as simple as possible. This is true of pretty much all programming, but I find to be especially so with Razor templates.

This section covers some of the more common C# constructs you can use. If you find you need to achieve something a bit more exotic, refer to the Razor syntax documentation at https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor.

7.3.1 Using C# in Razor templates

One of the most common requirements when working with Razor templates is to render a value you've calculated in C# to the HTML. For example, you might want to print the current year to use with a copyright statement in your HTML, to give

```
<p>Copyright 2020 ©</p>
```

or you might want to print the result of a calculation, for example

```
<p>The sum of 1 and 2 is <i>3</i><p>
```

You can do this in two ways, depending on the exact C# code you need to execute. If the code is a single statement, then you can use the `@` symbol to indicate you want to write the result to the HTML output, as shown in figure 7.4. You've already seen this used to write out values from the `PageModel` or from `ViewData`.

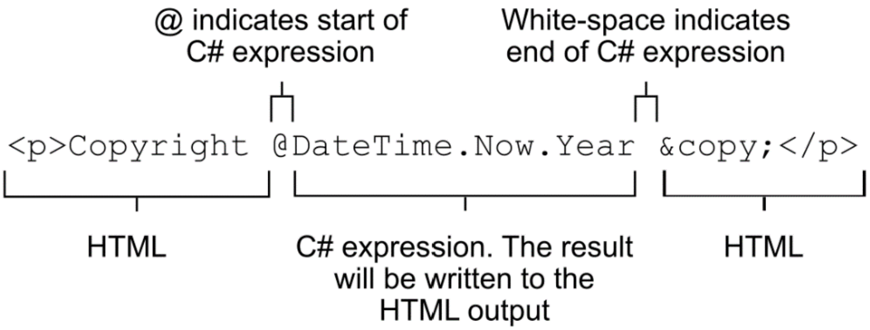

@ indicates start of C# expression
White-space indicates end of C# expression
<p>Copyright @DateTime.Now.Year &copy;</p>
HTML
C# expression. The result will be written to the HTML output
HTML

Figure 7.4 Writing the result of a C# expression to HTML. The `@` symbol indicates where the C# code begins and the expression ends at the end of the statement, in this case at the space.

If the C# you want to execute is something that *needs* a space, then you need to use parentheses to demarcate the C#, as shown in figure 7.5.

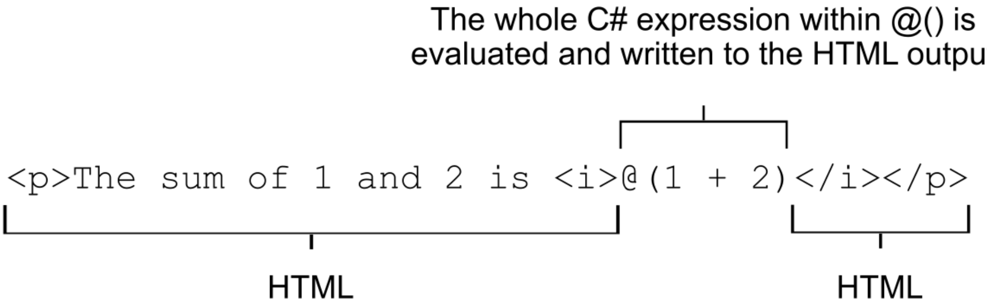

The whole C# expression within @() is evaluated and written to the HTML output
<p>The sum of 1 and 2 is <i>@(1 + 2)</i></p>
HTML
HTML

Figure 7.5 When a C# expression contains whitespace, you must wrap it in parentheses using `@()` so the Razor engine knows where the C# stops and HTML begins.

These two approaches, in which C# is evaluated and written directly to the HTML output, are called *Razor expressions*. Sometimes you want to execute some C#, but you don't need to output the values. We used this technique when we were setting values in `ViewData`:

```
@{
    ViewData["Title"] = "Home Page";
}
```

This example demonstrates a *Razor code block*, which is normal C# code, identified by the `@{}` structure. Nothing is written to the HTML output here; it's all compiled as though you'd written it in any other normal C# file.

TIP When you execute code within code blocks, it must be valid C#, so you need to add semicolons. Conversely, when you're writing values directly to the response using Razor expressions, you don't need them. If your output HTML breaks unexpectedly, keep an eye out for missing or rogue extra semicolons!

Razor expressions are one of the most common ways of writing data from your `PageModel` to the HTML output. You'll see the other approach, using Tag Helpers, in the next chapter. Razor's capabilities extend far further than this, however, as you'll see in the next section where you'll learn how to include traditional C# structures in your templates.

7.3.2 Adding loops and conditionals to Razor templates

One of the biggest advantages of using Razor templates over static HTML is the ability to dynamically generate the output. Being able to write values from your `PageModel` to the HTML using Razor expressions is a key part of that, but another common use is loops and conditionals. With these, you could hide sections of the UI, or produce HTML for every item in a list, for example.

Loops and conditionals include constructs such as `if` and `for` loops. Using them in Razor templates is almost identical to C#, but you need to prefix their usage with the `@` symbol. In case you're not getting the hang of Razor yet, when in doubt, throw in another `@`!

One of the big advantages of Razor in the context of ASP.NET Core is that it uses languages you're already familiar with: C# and HTML. There's no need to learn a whole new set of primitives for some other templating language: it's the same `if`, `foreach`, and `while` constructs you already know. And when you don't need them, you're writing raw HTML, so you can see exactly what the user will be getting in their browser.

In listing 7.6, I've applied a number of these different techniques in the template for displaying a to-do item. The `PageModel` has a `bool IsComplete` property, as well as a `List<string>` property called `Tasks`, which contains any outstanding tasks.

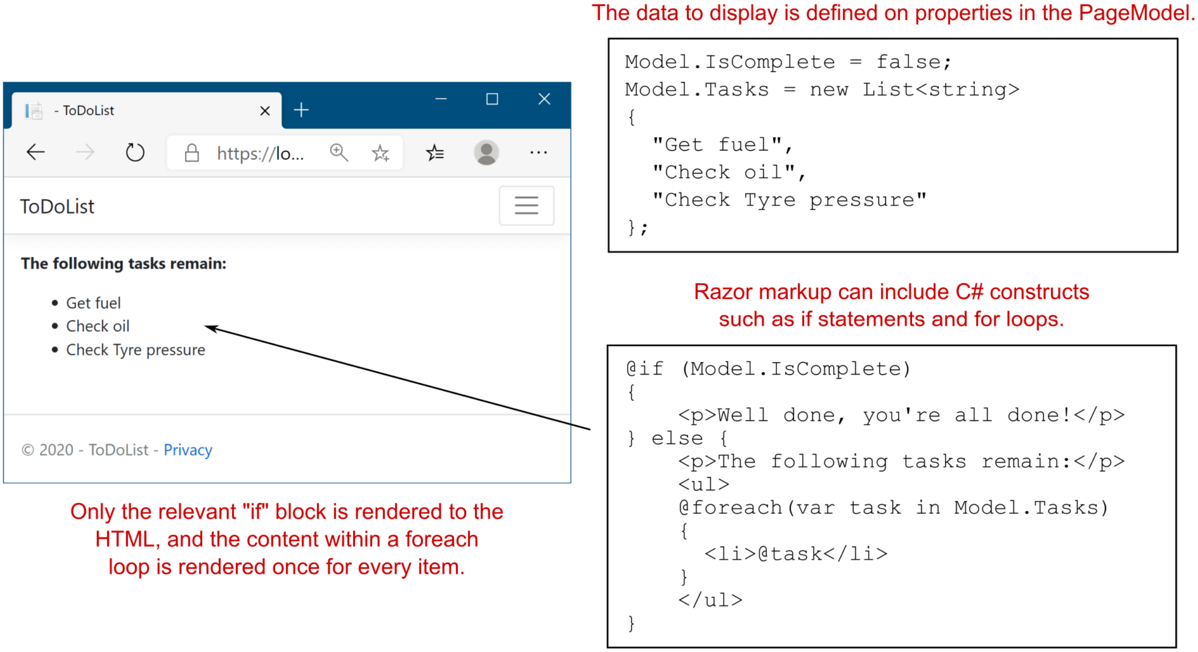
Listing 7.6 Razor template for rendering a `ToDoItemViewModel`

```
@page
@model ToDoItemModel                                           #A
<div>
    @if (Model.IsComplete)
    {                                                          #B
        <strong>Well done, you're all done!</strong>           #B
    }                                                          #B
    else
    {
        <strong>The following tasks remain:</strong>
        <ul>
            @foreach (var task in Model.Tasks)                #C
            {
                <li>@task</li>                                #D
            }
        </ul>
    }
</div>
```

#A The @model directive indicates the type of PageModel in Model.
#B The if control structure checks the value of the PageModel's IsComplete property at runtime.
#C The foreach structure will generate the <li> elements once for each task in Model.Tasks.
#D A razor expression is used to write the task to the HTML output.

This code definitely lives up to the promise of mixing C# and HTML! There are traditional C# control structures, like `if` and `foreach`, that you'd expect in any normal program, interspersed with the HTML markup that you want to send to the browser. As you can see, the `@` symbol is used to indicate when you're starting a control statement, but you generally let the Razor template infer when you're switching back and forth between HTML and C#.

The template shows how to generate dynamic HTML at runtime, depending on the exact data provided. If the model has outstanding `Tasks`, then the HTML will generate a list item for each task, producing output something like that shown in figure 7.6.



The data to display is defined on properties in the PageModel.

```
Model.IsComplete = false;
Model.Tasks = new List<string>
{
  "Get fuel",
  "Check oil",
  "Check Tyre pressure"
};
```

The following tasks remain:

- Get fuel
- Check oil
- Check Tyre pressure

Razor markup can include C# constructs such as if statements and for loops.

```
@if (Model.IsComplete)
{
    <p>Well done, you're all done!</p>
} else {
    <p>The following tasks remain:</p>
    <ul>
    @foreach(var task in Model.Tasks)
    {
      <li>@task</li>
    }
    </ul>
}
```

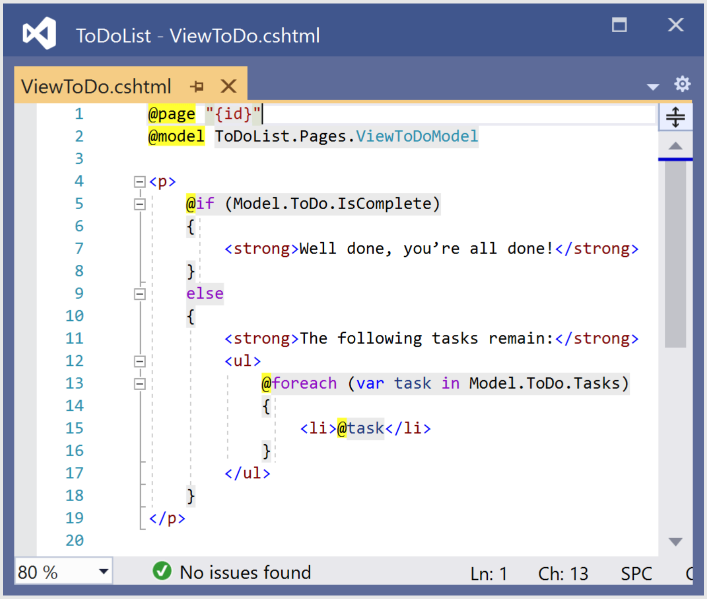
Only the relevant "if" block is rendered to the HTML, and the content within a foreach loop is rendered once for every item.

Figure 7.6 The Razor template generates a `<li>` item for each remaining task, depending on the data passed to the view at runtime. You can use an `if` block to render completely different HTML depending on the values in your model.

IntelliSense and tooling support

The mixture of C# and HTML might seem hard to read in the book, and that's a reasonable complaint. It's also another valid argument for trying to keep your Razor templates as simple as possible.

Luckily, if you’re using an editor like Visual Studio or Visual Studio Code, then the tooling can help somewhat. As you can see in this figure, the C# portions of the code are shaded to help distinguish them from the surrounding HTML.



```
@page "{id}"
@model ToDoList.Pages.ViewToDoModel

<p>
    @if (Model.ToDo.IsComplete)
    {
        <strong>Well done, you’re all done!</strong>
    }
    else
    {
        <strong>The following tasks remain:</strong>
        <ul>
            @foreach (var task in Model.ToDo.Tasks)
            {
                <li>@task</li>
            }
        </ul>
    }
</p>

```

Visual Studio shades the C# regions of code and highlights @ symbols where C# transitions to HTML. This makes the Razor templates easier to read.

Although the ability to use loops and conditionals is powerful—they’re one of the advantages of Razor over static HTML—they also add to the complexity of your view. Try to limit the amount of logic in your views to make them as easy to understand and maintain as possible.

A common trope of the ASP.NET Core team is that they try to ensure you “fall into the pit of success” when building an application. This refers to the idea that, by default, the *easiest* way to do something should be the *correct* way of doing it. This is a great philosophy, as it means you shouldn’t get burned by, for example, security problems if you follow the standard approaches. Occasionally, however, you may need to step beyond the safety rails; a common use case is when you need to render some HTML contained in a C# object to the output, as you’ll see in the next section.

7.3.3 Rendering HTML with Raw

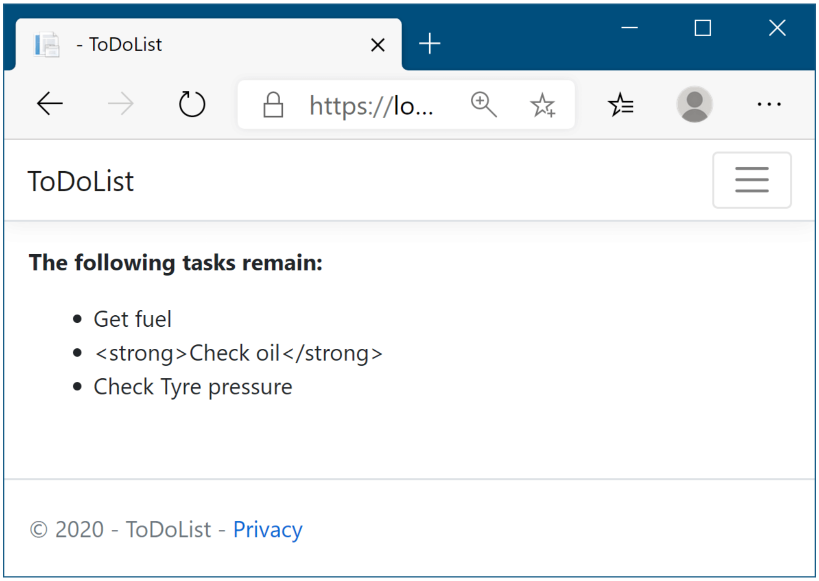
In the previous example, we rendered the list of tasks to HTML by writing the `string task` using the `@task` Razor expression. But what if the `task` variable contains HTML you want to display, so instead of `"Check oil"` it contains `"<strong>Check oil</strong>"`? If you use a Razor expression to output this as you did previously, then you might hope to get this:

```
<li><strong>Check oil</strong></li>
```

But that’s not the case. The HTML generated comes out like this:

```
<li>&lt;strong&gt;Check oil&lt;/strong&gt;</li>
```

Hmm, looks odd, right? What’s happened here? Why did the template not write your variable to the HTML, like it has in previous examples? If you look at how a browser displays this HTML, like in figure 7.7, then hopefully it makes more sense.



ToDoList

The following tasks remain:

- Get fuel
- <strong>Check oil</strong>
- Check Tyre pressure

© 2020 - ToDoList - Privacy

Figure 7.7 The second item, <strong>Check oil<strong> has been HTML-encoded, so the `<strong>` elements are visible to the user as part of the task. This avoids any security issues, as users can’t inject malicious scripts into your HTML.

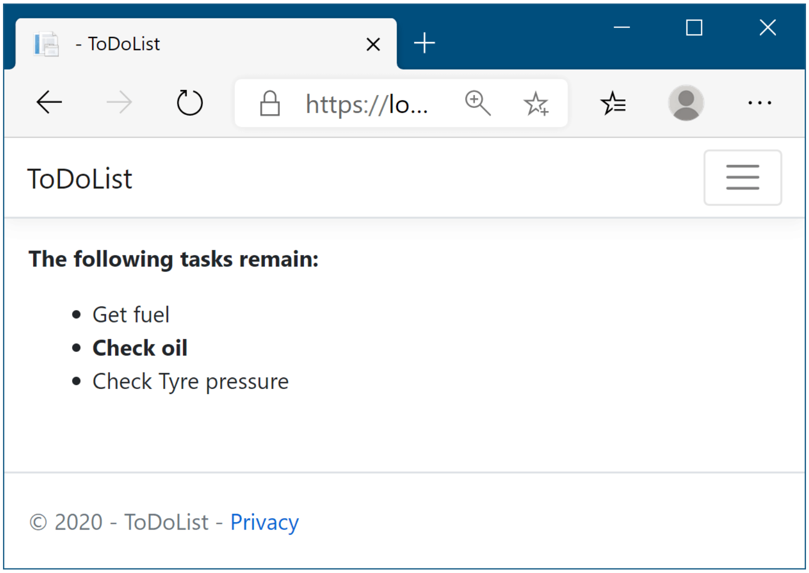
Razor templates encode C# expressions before they’re written to the output stream. This is primarily for security reasons; writing out arbitrary strings to your HTML could allow users to inject malicious data and JavaScript into your website. Consequently, the C# variables you print in your Razor template get written as HTML-encoded values.

In some cases, you might need to directly write out HTML contained in a `string` to the response. If you find yourself in this situation, first, stop. Do you *really* need to do this? If the values you’re writing have been entered by a user, or were created based on values provided by users, then there’s a serious risk of creating a security hole in your website.

If you *really* need to write the variable out to the HTML stream, then you can do so using the `Html` property on the view page and calling the `Raw` method:

```
<li>@Html.Raw(task)</li>
```

With this approach, the string in `task` will be directly written to the output stream, producing the HTML you originally wanted, `<li><strong>Check oil</strong></li>`, which renders as shown in figure 7.8.



ToDoList

The following tasks remain:

- Get fuel
- **Check oil**
- Check Tyre pressure

© 2020 - ToDoList - Privacy

Figure 7.8 The second item, "`<strong>Check oil<strong>`" has been output using `Html.Raw()`, so it hasn't been HTML encoded. The `<strong>` elements result in the second item being shown in bold instead. Using `Html.Raw()` in this way should be avoided where possible, as it is a security risk.

WARNING **Using `Html.Raw` on user input creates a security risk that users could use to inject malicious code into your website. Avoid using `Html.Raw` if possible!**

The C# constructs shown in this section can be useful, but they can make your templates harder to read. It's generally easier to understand the intention of Razor templates that are predominantly HTML markup rather than C#.

In the previous version of ASP.NET, these constructs, and in particular the `Html` helper property, were the standard way to generate dynamic markup. You can still use this approach in ASP.NET Core by using the various `HtmlHelper`[29] methods on the `Html` property, but these have largely been superseded by a cleaner technique: Tag Helpers.

NOTE **I'll discuss Tag Helpers, and how to use them to build HTML forms, in the next chapter.**

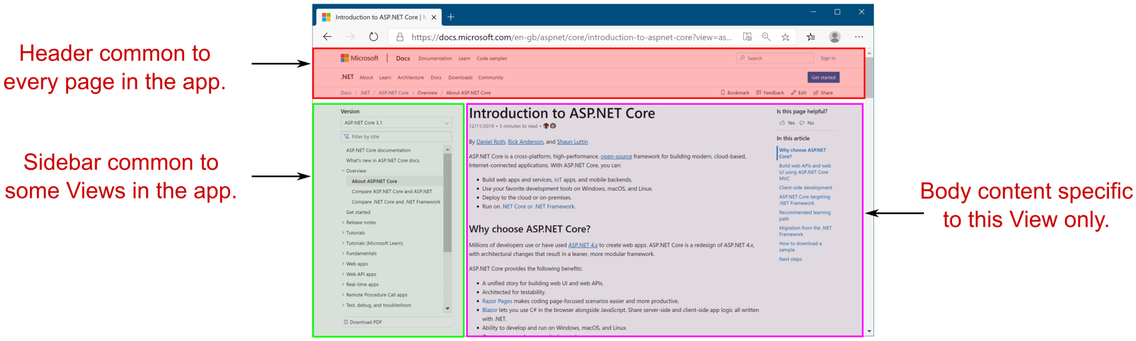
[29] HTML Helpers are almost obsolete, though they're still available if you prefer to use them.

Tag Helpers are a useful feature that's new to Razor in ASP.NET Core, but a number of other features have been carried through from the previous version of ASP.NET. In the next section of this chapter, you'll see how you can create nested Razor templates and use partial views to reduce the amount of duplication in your views.

7.4 Layouts, partial views, and _ViewStart

In this section you'll learn about layouts and partial views, which allow you to extract common code to reduce duplication. These files make it easier to make changes to your HTML that affect multiple pages at once. You'll also learn how to run common code for every Razor Page using _ViewStart and _ViewImports, and how to include optional sections in your pages.

Every HTML document has a certain number of elements that are required: `<html>`, `<head>`, and `<body>`. As well, there are often common sections that are repeated on every page of your application, such as the header and footer, as shown in figure 7.9. Each page on your application will also probably reference the same CSS and JavaScript files.



Header common to every page in the app.

Sidebar common to some Views in the app.

Body content specific to this View only.

Introduction to ASP.NET Core | M
https://docs.microsoft.com/en-gb/aspnet/core/introduction-to-aspnet-core?view=as...
Microsoft
Docs
Documentation
Learn
Code samples
Search
Sign in
.NET
About
Learn
Architecture
Docs
Downloads
Community
Get started
Docs / .NET / ASP.NET Core / Overview / About ASP.NET Core
Bookmark
Feedback
Edit
Share
Version
ASP.NET Core 3.1
Filter by title
ASP.NET Core documentation
What's new in ASP.NET Core docs
Overview
About ASP.NET Core
Compare ASP.NET Core and ASP.NET
Compare .NET Core and .NET Framework
Get started
Release notes
Tutorials
Tutorials (Microsoft Learn)
Fundamentals
Web apps
Web API apps
Real-time apps
Remote Procedure Call apps
Test, debug, and troubleshoot
Download PDF
Is this page helpful?
Yes
No
In this article
Why choose ASP.NET Core?
Build web APIs and web UI using ASP.NET Core MVC
Client-side development
ASP.NET Core targeting .NET Framework
Recommended learning path
Migration from the .NET Framework
How to download a sample
Next steps

Introduction to ASP.NET Core

12/11/2019 • 5 minutes to read •

By Daniel Roth, Rick Anderson, and Shaun Luttin

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, Internet-connected applications. With ASP.NET Core, you can:

- Build web apps and services, IoT apps, and mobile backends.
- Use your favorite development tools on Windows, macOS, and Linux.
- Deploy to the cloud or on-premises.
- Run on .NET Core or .NET Framework.

Why choose ASP.NET Core?

Millions of developers use or have used ASP.NET 4.x to create web apps. ASP.NET Core is a redesign of ASP.NET 4.x, with architectural changes that result in a leaner, more modular framework.

ASP.NET Core provides the following benefits:

- A unified story for building web UI and web APIs.
- Architected for testability.
- Razor Pages makes coding page-focused scenarios easier and more productive.
- Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
- Ability to develop and run on Windows, macOS, and Linux.

Figure 7.9 A typical web application has a block-based layout, where some blocks are common to every page of your application. The header block will likely be identical across your whole application, but the sidebar may only be identical for the pages in one section. The body content will differ for every page in your application.

All these different elements add up to a maintenance nightmare. If you had to manually include these in every view, then making any changes would be a laborious, error-prone process or manually editing every page. Instead, Razor lets you extract these common elements into *layouts*.

DEFINITION **A *layout* in Razor is a template that includes common code. It can't be rendered directly, but it can be rendered in conjunction with normal Razor views.**

By extracting your common markup into layouts, you can reduce the duplication in your app. This makes changes easier, makes your views easier to manage and maintain, and is generally good practice!

7.4.1 Using layouts for shared markup

Layout files are, for the most part, normal Razor templates that contain markup common to more than one page. An ASP.NET Core app can have multiple layouts, and layouts can reference other layouts. A common use for this is to have different layouts for different sections of your application. For example, an e-commerce website might use a three-column view for most pages, but a single-column layout when you come to the checkout pages, as shown in figure 7.10.

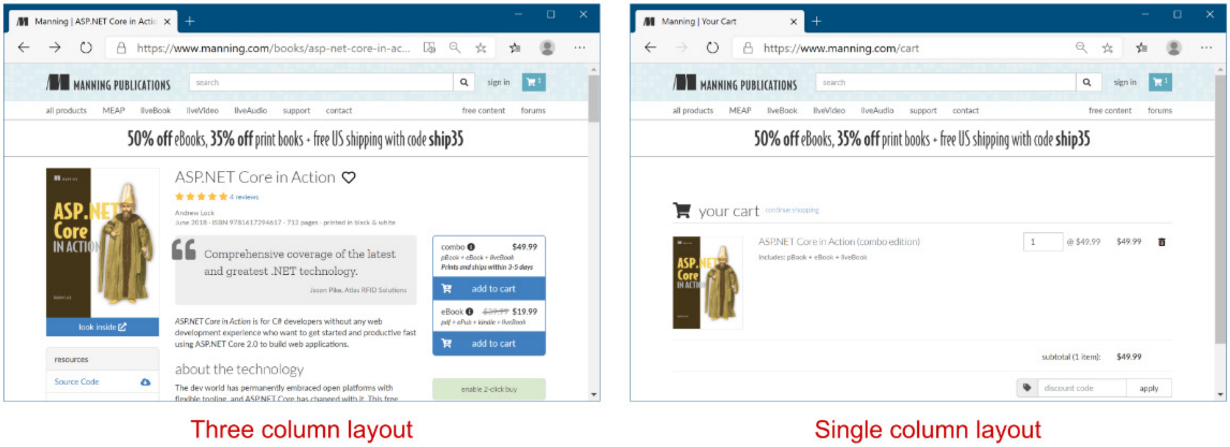

Manning | ASP.NET Core in Actio
https://www.manning.com/books/asp-net-core-in-ac...
MANNING PUBLICATIONS
50% off eBooks, 35% off print books + free US shipping with code ship35
ASP.NET Core in Action
Comprehensive coverage of the latest and greatest .NET technology.
combo $49.99
add to cart
eBook $39.99 $19.99
add to cart
look inside
resources
Source Code
about the technology
Three column layout
Manning | Your Cart
https://www.manning.com/cart
MANNING PUBLICATIONS
50% off eBooks, 35% off print books + free US shipping with code ship35
your cart
ASP.NET Core in Action (combo edition)
@ $49.99
$49.99
subtotal (1 item):
$49.99
discount code
apply
Single column layout

Figure 7.10 The https://manning.com website uses different layouts for different parts of the web application. The product pages use a three-column layout, but the cart page uses a single-column layout.

You'll often use layouts across many different Razor Pages, so they're typically placed in the Pages/Shared folder. You can name them anything you like, but there's a common convention to use _Layout.cshtml as the filename for the base layout in your application. This is the default name used by the Razor Page templates in Visual Studio and the .NET CLI.

> **TIP** A common convention is to prefix your layout files with an underscore (_) to distinguish them from standard Razor templates in your Pages folder.

A layout file looks similar to a normal Razor template, with one exception: every layout must call the `@RenderBody()` function. This tells the templating engine where to insert the content from the child views. A simple layout is shown in the following listing. Typically, your application will reference all your CSS and JavaScript files in the layout, as well as include all the common elements such as headers and footers, but this example includes pretty much the bare minimum HTML.

Listing 7.7 A basic Layout.cshtml file calling `RenderBody`

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewData["Title"]</title>                 #A
    <link rel="stylesheet" href="~/css/site.css" />   #B
</head>
<body>
    @RenderBody()                                     #C
</body>
</html>
```

#A ViewData is the standard mechanism for passing data to a layout from a view.
#B Elements common to every page, such as your CSS, are typically found in the layout.
#C Tells the templating engine where to insert the child view's content

As you can see, the layout file includes the required elements, such as `<html>` and `<head>`, as well as elements you need on every page, such as `<title>` and `<link>`. This example also shows the benefit of storing the page title in `ViewData`; the layout can render it in the `<title>` element so that it shows in the browser's tab, as shown in figure 7.11.

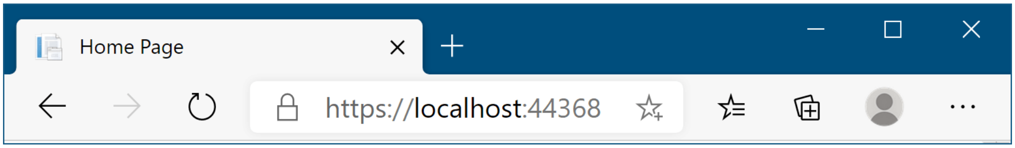


Figure 7.11 The contents of the `<title>` element is used to name the tab in the user's browser, in this case Home Page.

Views can specify a layout file to use by setting the `Layout` property inside a Razor code block.

Listing 7.8 Setting the `Layout` property from a view

```
@{
    Layout = "_Layout";                  #A
    ViewData["Title"] = "Home Page";     #B
}
<h1>@ViewData["Title"]</h1>              #C
<p>This is the home page</p>             #C
```

#A Set the layout for the page to _Layout.cshtml.
#B ViewData is a convenient way of passing data from a Razor view to the layout.
#C The content in the Razor view to render inside the layout

Any contents in the view will be rendered inside the layout, where the call to `@RenderBody()` occurs. Combining the two previous listings will result in the following HTML being generated and sent to the user.

Listing 7.9 Rendered output from combining a view with its layout

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Home Page</title>                           #A
    <link rel="stylesheet" href="/css/site.css" />
</head>
<body>
    <h1>Home Page</h1>                                  #B
    <p>This is the home page</p>                           #B
</body>
<html>
```

#A ViewData set in the view is used to render the layout.
#B The RenderBody call renders the contents of the view.

Judicious use of layouts can be extremely useful in reducing the duplication on a page. By default, layouts only provide a single location where you can render content from the view, at the call to `@RenderBody`. In cases where this is too restrictive, you can render content using *sections*.

7.4.2 Overriding parent layouts using sections

A common requirement when you start using multiple layouts in your application is to be able to render content from child views in more than one place in your layout. Consider the case of a layout that uses two columns. The view needs a mechanism for saying "render *this* content in the *left* column" and "render this *other* content in the *right* column". This is achieved using *sections*.

> **NOTE** Remember, all of the features outlined in this chapter are specific to Razor, which is a server-side rendering engine. If you're using a client-side SPA framework to build your application, you'll likely handle these requirements in other ways, either within the client code or by making multiple requests to a Web API endpoint.

Sections provide a way of organizing where view elements should be placed within a layout. They're defined in the view using an `@section` definition, as shown in the following listing, which defines the HTML content for a sidebar separate from the main content, in a section called `Sidebar`. The `@section` can be placed anywhere in the file, top or bottom, wherever is convenient.

Listing 7.10 Defining a section in a view template

```
@{
    Layout = "_TwoColumn";
}
@section Sidebar {                             #A
    <p>This is the sidebar content</p>         #A
}                                              #A
```

```
<p>This is the main content </p>                #B
```

#A All content inside the braces is part of the Sidebar section, not the main body content.
#B Any content not inside an @section will be rendered by the @RenderBody call.

The section is rendered in the parent layout with a call to `@RenderSection()`. This renders the content contained in the child section into the layout. Sections can be either required or optional. If they're required, then a view *must* declare the given `@section`; if they're optional then they can be omitted, and the layout will skip it. Skipped sections won't appear in the rendered HTML. This listing shows a layout that has a required section called `Sidebar`, and an optional section called `Scripts`.

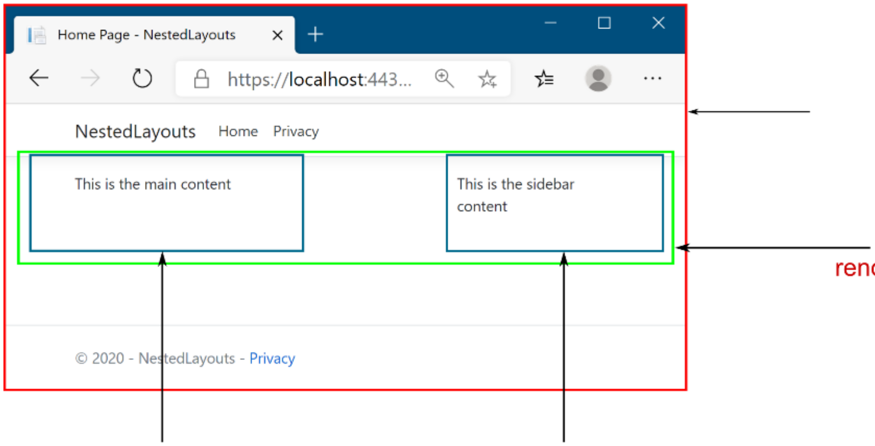
Listing 7.11 Rendering a section in a layout file, _TwoColumn.cshtml

```
@{
    Layout = "_Layout";                          #A
}
<div class="main-content">
    @RenderBody()                                #B
</div>
<div class="side-bar">
    @RenderSection("Sidebar", required: true)    #C
</div>
@RenderSection("Scripts", required: false)       #D
```

#A This layout is nested inside a layout itself.
#B Renders all the content from a view that isn't part of a section
#C Renders the Sidebar section; if the Sidebar section isn't defined in the view, throws an error
#D Renders the Scripts section; if the Scripts section isn't defined in the view, ignore it.

> **TIP** It's common to have an optional section called `Scripts` in your layout pages. This can be used to render additional JavaScript that's required by some views, but that isn't needed on every view. A common example is the jQuery Unobtrusive Validation scripts for client-side validation. If a view requires the scripts, it adds the appropriate `@section Scripts` to the Razor markup.

You may notice that the previous listing defines a `Layout` property, even though it's a layout itself, not a view. This is perfectly acceptable, and lets you create nested hierarchies of layouts, as shown in figure 7.12.


Home Page - NestedLayouts
https://localhost:443...
NestedLayouts
Home
Privacy
This is the main content
This is the sidebar content
© 2020 - NestedLayouts - Privacy

_Layout.cshtml defines the HTML in the Header and footer.

_TwoColumn.cshtml is rendered inside _Layout.cshtml.

The main content of the View is rendered in _TwoColumn.cshtml by RenderBody.

The sidebar content of the View is rendered in _TwoColumn.cshtml by RenderSection(Sidebar).

Figure 7.12 Multiple layouts can be nested to create complex hierarchies. This allows you to keep the elements common to all views in your base layout and extract layout common to multiple views into sub-layouts.

> **TIP** **Most websites these days need to be "responsive", so they work on a wide variety of devices. You generally *shouldn't* use layouts for this. Don't serve different layouts for a single page based on the device making the request. Instead, serve the same HTML to all devices, and use CSS on the client-side to adapt the display of your web page as required.**

Layout files and sections provide a lot of flexibility to build sophisticated UIs, but one of their most important uses is in reducing the duplication of code in your application. They're perfect for avoiding duplication of content that you'd need to write for every view. But what about those times when you find you want to reuse part of a view somewhere else? For those cases, you have partial views.

7.4.3 Using partial views to encapsulate markup

Partial views are exactly what they sound like—they're part of a view. They provide a means of breaking up a larger view into smaller, reusable chunks. This can be useful for both reducing the complexity in a large view by splitting it into multiple partial views, or for allowing you to reuse part of a view inside another.

Most web frameworks that use server-side rendering have this capability—Ruby on Rails has partial views, Django has inclusion tags, and Zend has partials. All of these work in the same way, extracting common code into small, reusable templates. Even client-side templating engines such as Mustache and Handlebars used by client-side frameworks like Angular and Ember have similar "partial view" concepts.

Consider a to-do list application again. You might find you have a Razor Page called ViewToDo.cshtml that displays a single to-do with a given `id`. Later on, you create a new Razor Page, RecentToDos.cshtml, that displays the five most recent to-do items. Instead of copying and pasting the code from one page to the other, you could create a partial view, called _ToDo.cshtml.

Listing 7.12 Partial view _ToDo.cshtml for displaying a `ToDoItemViewModel`

```
@model ToDoItemViewModel                    #A
<h2>@Model.Title</h2>                       #B
<ul>                                        #B
    @foreach (var task in Model.Tasks)      #B
    {                                       #B
        <li>@task</li>                      #B
    }                                       #B
</ul>                                       #B
```

#A Partial views can bind to data in the Model property, like a normal Razor Page uses a PageModel.
#B The content of the partial view, which previously existed in the ViewToDo.cshtml file

Partial views are a bit like Razor Pages without the `PageModel` and handlers. Partial views are purely about rendering small sections of HTML, rather than handling requests, model binding and validation, and calling the application model. They are great for encapsulating small usable bits of HTML that you need to generate on multiple Razor Pages.

Both the ViewToDo.cshtml and RecentToDos.cshtml Razor Pages can render the _ToDo .cshtml partial view, which handles generating the HTML for a single class. Partial views are rendered using the `<partial />` Tag Helper, providing the name of the partial view to render, and the data (the model) to render. For example, the RecentToDos.cshtml view could achieve this as shown in the following listing.

Listing 7.13 Rendering a partial view from a Razor Page

```
@page                                            #A
@model RecentToDoListModel                       #B

@foreach(var todo in Model.RecentItems)          #C
{
    <partial name="_ToDo" model="todo" />        #D
}
```

#A This is a Razor Page, so it uses the @page directive. Partial views do not use @page.
#B The PageModel contains the list of recent items to render
#C Loop through the recent items. todo is a ToDoItemViewModel, as required by the partial view.
#D Use the partial tag helper to render the _ToDo partial view, passing in the model to render.

When you render a partial view without providing an absolute path or file extension, for example _ToDo in listing 7.13, the framework tries to locate the view by searching the Pages folder, starting from the Razor Page that invoked it. For example, if your Razor Page is located at Pages/Agenda/ToDos/RecentToDos.chstml, the framework would look in the following places for a file called _ToDo.chstml:

- Pages/Agenda/ToDos/ (the current Razor Page's folder)
- Pages/Agenda/
- Pages/
- Pages/Shared/
- Views/Shared/

The first location that contains a file called _ToDo.cshtml will be selected. If you include the cshtml file extension when you reference the partial view, the framework will *only* look in the current Razor Page's folder. Also, if you provide an absolute path to the partial, such as /Pages/Agenda/ToDo.cshtml, then that's the only place the framework will look.[30]

NOTE Like layouts, partial views are typically named with a leading underscore.

The Razor code contained in a partial view is almost identical to a standard view. The main difference is the fact that partial views are only called from other views. The other difference is that partial views don't run _ViewStart.cshtml when they execute, which you'll see shortly.

Child actions in ASP.NET Core

In the previous version of ASP.NET MVC, there was the concept of a *child action*. This was an action method that could be invoked *from inside a view*. This was the main mechanism for rendering discrete sections of a complex layout that had nothing to do with the main action method. For example, a child action method might render the shopping cart on an e-commerce site.

This approach meant you didn't have to pollute every page's view model with the view model items required to render the shopping cart, but it fundamentally broke the MVC design pattern, by referencing controllers from a view.

In ASP.NET Core, child actions are no more. *View components* have replaced them. These are conceptually quite similar in that they allow both the execution of arbitrary code and the rendering of HTML, but they don't directly invoke controller actions. You can think of them as a more powerful partial view that you should use anywhere a partial view needs to contain significant code or business logic. You'll see how to build a small view component in chapter 19.

Partial views aren't the only way to reduce duplication in your view templates. Razor also allows you to pull common elements such as namespace declarations and layout configuration into centralized files. In the next section, you'll see how to wield these files to clean up your templates.

[30] As with most of Razor Pages, the search locations are conventions that you can customize if you wish. If you find the need, you can customize the paths as shown here https://www.learnrazorpages.com/razor-pages/partial-pages#naming-and-locating-partial-pages.

7.4.4 Running code on every view with _ViewStart and _ViewImports

Due to the nature of views, you'll inevitably find yourself writing certain things repeatedly. If all of your views use the same layout, then adding the following code to the top of every page feels a little redundant:

```
@{
    Layout = "_Layout";
}
```

Similarly, if you find you need to reference objects from a different namespace in your Razor views, then having to add `@using WebApplication1.Models` to the top of every page can get to be a chore. Thankfully, ASP.NET Core includes two mechanisms for handling these common tasks: `_ViewImports.cshtml` and `_ViewStart.cshtml`.

IMPORTING COMMON DIRECTIVES WITH _VIEWIMPORTS

The _ViewImports.cshtml file contains directives that will be inserted at the top of every view. This includes things like the `@using` and `@model` statements that you've already seen—basically any Razor directive. To avoid adding a using statement to every view, you can include it in here instead.

Listing 7.13 A typical _ViewImports.cshtml file importing additional namespaces

```
@using WebApplication1                                       #A
@using WebApplication1.Pages                                 #A
@using WebApplication1.Models                                   #B
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers         #C
```

#A The default namespace of your application and the Pages folder
#B Add this directive to avoid placing it in every view
#C Makes Tag Helpers available in your views, added by default

The _ViewImports.cshtml file can be placed in any folder, and it will apply to all views and sub-folders in that folder. Typically, it's placed in the root Pages folder so that it applies to every Razor Page and partial view in your app.

It's important to note that you should *only* put Razor directives in _ViewImports.cshtml—you can't put any old C# in there. As you can see in the previous listing, this is limited to things like `@using` or the `@addTagHelper` directive that you'll learn about in the next chapter. If you want to run some arbitrary C# at the start of every view in your application, for example to set the `Layout` property, then you should use the _ViewStart.cshtml file instead.

RUNNING CODE FOR EVERY VIEW WITH _VIEWSTART

You can easily run common code at the start of every Razor Page by adding a _ViewStart.cshtml file to the Pages folder in your application. This file can contain any Razor code, but it's typically used to set the `Layout` for all the pages in your application, as shown in the following listing. You can then omit the `Layout` statement from all pages that use the default layout. If a view needs to use a nondefault layout then you can override it by setting the value in the Razor page itself.

Listing 7.14 A typical _ViewStart.cshtml file setting the default layout

```
@{
    Layout = "_Layout";
}
```

Any code in the _ViewStart.cshtml file runs before the view executes. Note that _ViewStart.cshtml only runs for Razor Page views—it doesn't run for layouts or partials views. Also, note that the names for these special Razor files are enforced rather than conventions you can change.

> **WARNING** You must use the names _ViewStart.cshtml and _ViewImports .cshtml for the Razor engine to locate and execute them correctly. To apply them to all your app's pages, add them to the root of the Pages folder, not to the Shared subfolder.

You can specify additional _ViewStart.cshtml or _ViewImports.cshtml files to run for a subset of your views by including them in a subfolder in Pages. The files in the subfolders will run after the files in the root Pages folder.

Partial views, layouts, and AJAX

This chapter describes using Razor to render full HTML pages server-side, which are then sent to the user's browser in traditional web apps. A common alternative approach when building web apps is to use a JavaScript client-side framework to build a Single Page Application (SPA), which renders the HTML client-side in the browser.

One of the technologies SPAs typically use is AJAX (Asynchronous JavaScript and XML), in which the browser sends requests to your ASP.NET Core app without reloading a whole new page. It's also possible to use AJAX requests with apps that use server-side rendering. To do so, you'd use JavaScript to request an update for part of a page.

If you want to use AJAX with an app that uses Razor, you should consider making extensive use of partial views. You can then expose these via additional Razor Page handlers, as shown in this article https://www.learnrazorpages.com/razor-pages/ajax/partial-update. Using AJAX can reduce the overall amount of data that needs to be sent back and forth between the browser and your app, and it can make your app feel smoother and more responsive, as it requires fewer full-page loads. But using AJAX with Razor can add complexity, especially for larger apps. If you foresee yourself making extensive use of AJAX to build a highly dynamic web app, you might want to consider using Web API controllers with a client-side framework (see chapter 9), or consider Blazor instead.

In this chapter I've focused on using Razor views with the Razor Page framework, as that's the approach I suggest if you're creating a server-side rendered ASP.NET Core application. However, as I described in chapter 4, you may want to use MVC controllers in some cases. In the final section of this chapter we look at how you can render Razor views from your MVC controller actions, and how the framework locates the correct Razor view to render.

7.5 Selecting a view from an MVC controller

This section covers:

- How MVC controllers use `ViewResult`s to render Razor views
- How to create a new Razor view
- How the framework locates a Razor view to render

If you follow my advice from chapter 4, then you should be using Razor Pages for your server-side rendered applications instead of the MVC controllers that were common in versions 1.x and 2.0. One of the big advantages Razor Pages gives is the close coupling of a Razor view to the associated page handlers, instead of having to navigate between multiple folders in your solution.

If for some reason you *do* need to use MVC controllers instead of Razor Pages, then it’s important to understand how you choose which view to render once an action method has executed. Figure 7.13 shows a zoomed-in view of this process, right after the action has invoked the application model and received some data back.

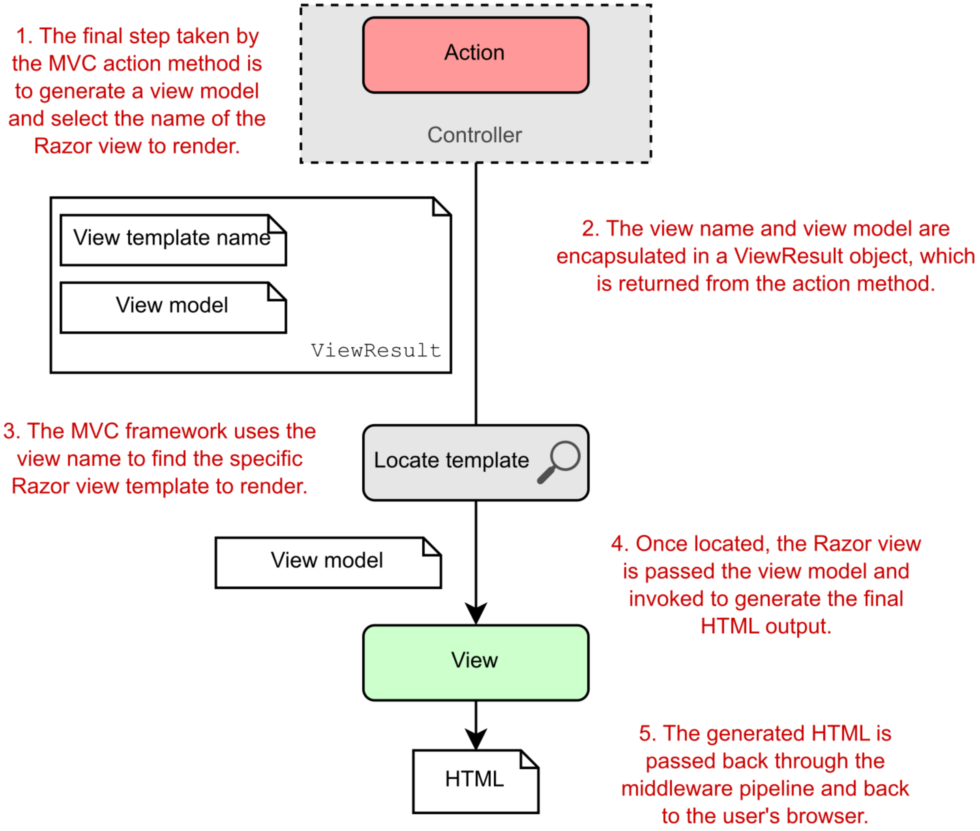

1. The final step taken by the MVC action method is to generate a view model and select the name of the Razor view to render.
Action
Controller
View template name
View model
ViewResult
2. The view name and view model are encapsulated in a ViewResult object, which is returned from the action method.
3. The MVC framework uses the view name to find the specific Razor view template to render.
Locate template
View model
4. Once located, the Razor view is passed the view model and invoked to generate the final HTML output.
View
HTML
5. The generated HTML is passed back through the middleware pipeline and back to the user's browser.

Figure 7.13 The process of generating HTML from an MVC controller using a ViewResult. This is very similar to the process for a Razor Page. The main difference is that for Razor Pages, the view is an integral part of the Razor Page; for MVC controllers, the view must be located at runtime.

Some of this figure should be familiar—it's the lower half of figure 4.7 from chapter 4 (with a couple of additions) and is the MVC equivalent of figure 7.1. It shows that the MVC controller action method uses a `ViewResult` object to indicate that a Razor view should be rendered. This `ViewResult` contains the name of the Razor view template to render and a view model, an arbitrary POCO class containing the data to render.

> **NOTE** I discussed `ViewResult`s in chapter 4. They are the MVC equivalent of Razor Page's `PageResult`. The main difference is that a `ViewResult` includes a view name to render and a model to pass to the view template, while a `PageResult` always renders the Razor Page's associated view and passes the `PageModel` to the view template.

After returning a ViewResult from an action method, the control flow passes back to the MVC framework, which uses a series of heuristics to locate the view, based on the template name provided. Once a Razor view template has been located, the Razor engine passes the view model from the ViewResult to the view and executes the template to generate the final HTML. This final step, rendering the HTML, is essentially the same process as for Razor Pages.

You saw how to create controllers in chapter 4, and in this section, you'll see how to create views and ViewResult objects and how to specify the template to render. You can add a new view template to your application in Visual Studio by right-clicking in an MVC application in Solution Explorer and choosing Add > New Item, and selecting Razor View from the dialog, as shown in figure 7.14. If you aren't using Visual Studio, create a blank new file in the Views folder with the file extension .cshtml.

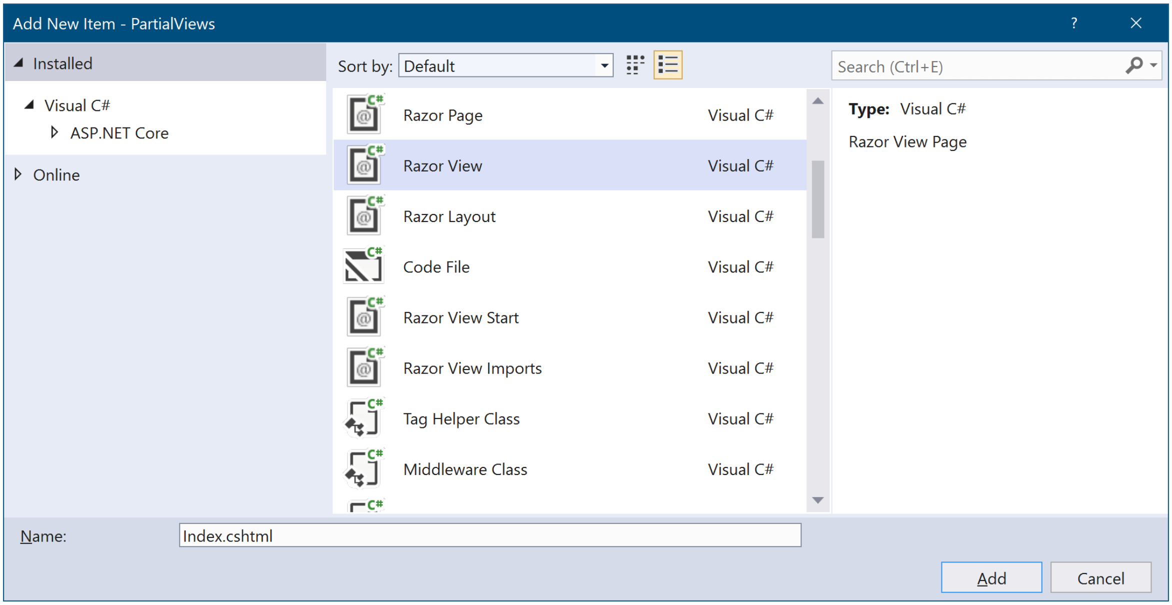


Figure 7.14 The Add New Item dialog. Choosing Razor View will add a new Razor view template file to your application.

With your view template created, you now need to invoke it. In most cases, you won't create a ViewResult directly in your action methods. Instead, you'll use one of the View helper methods on the Controller base class. These helper methods simplify passing in a view model and selecting a view template, but there's nothing magic about them—all they do is create ViewResult objects.

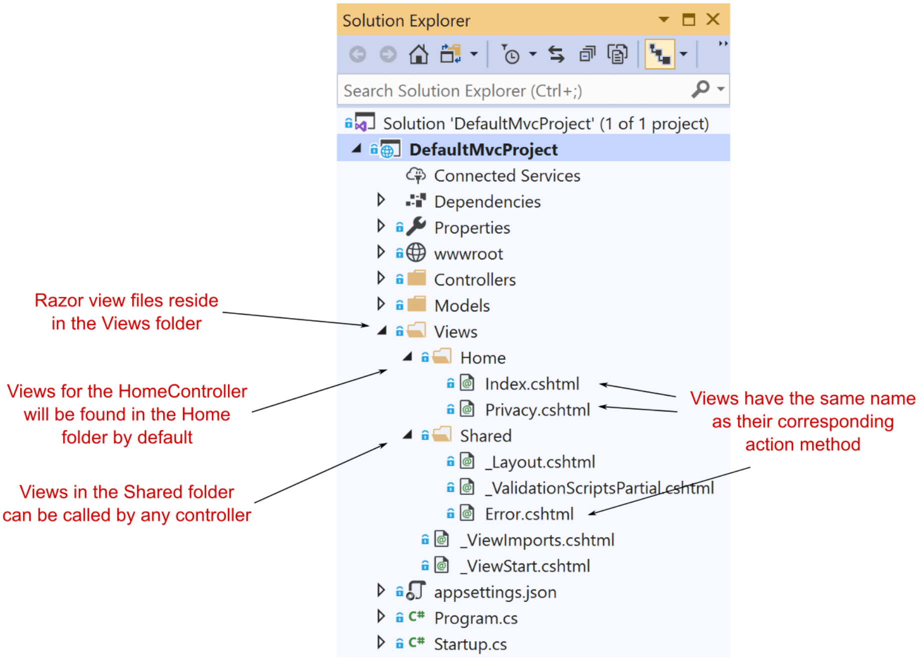
In the simplest case, you can call the View method without any arguments, as shown in the listing below. This helper method returns a ViewResult that will use conventions to find the view template to render, and will not supply a view model when executing the view.

Listing 7.15 Returning `ViewResult` from an action method using default conventions

```
public class HomeController : Controller          #A
{
    public IActionResult Index()
    {
        return View();                               #B
    }
}
```

#A Inheriting from the Controller base class makes the View helper methods available.
#B The View helper method returns a ViewResult.

In this example, the `View` helper method returns a `ViewResult` without specifying the name of a template to run. Instead, the name of the template to use is based on the name of the controller and the name of the action method. Given that the controller is called `HomeController` and the method is called `Index`, by default the Razor template engine looks for a template at the Views/Home/Index.cshtml location, as shown in figure 7.15.



Razor view files reside in the Views folder

Views for the HomeController will be found in the Home folder by default

Views in the Shared folder can be called by any controller

Views have the same name as their corresponding action method

Figure 7.15 View files are located at runtime based on naming conventions. Razor view files reside in a folder based on the name of the associated MVC controller and are named with the name of the action method that requested it. Views in the Shared folder can be used by any controller.

This is another case of using conventions in MVC to reduce the amount of boilerplate you have to write. As always, the conventions are optional. You can also explicitly pass the name of the template to run as a `string` to the `View` method. For example, if the `Index` method instead returned `View("ListView")`, then the templating engine would look for a template called ListView.cshtml instead. You can even specify the complete path to the view file, relative to your application's root folder, such as `View("Views/global.cshtml")`, which would look for the template at the Views/global.chtml location.

> **NOTE** When specifying the absolute path to a view, you must include both the top-level Views folder and the cshtml file extension in the path. This is similar to the rules for locating partial view templates.

The process of locating an MVC Razor view is very similar to the process of locating a partial view to render, as you saw in section 7.4. The framework searches in multiple locations to find the requested view. The difference is that for Razor Pages the search process only happens for *partial* view rendering, as the main Razor view to render is already known—it's the Razor Page's view template.

Figure 7.16 shows the complete process used by the MVC framework to locate the correct View template to execute when a `ViewResult` is returned from an MVC controller. It's possible for more than one template to be eligible, for example if an Index.chstml file exists in both the Home and Shared folders. Similar to the rules for locating partial views, the engine will use the first template it finds.

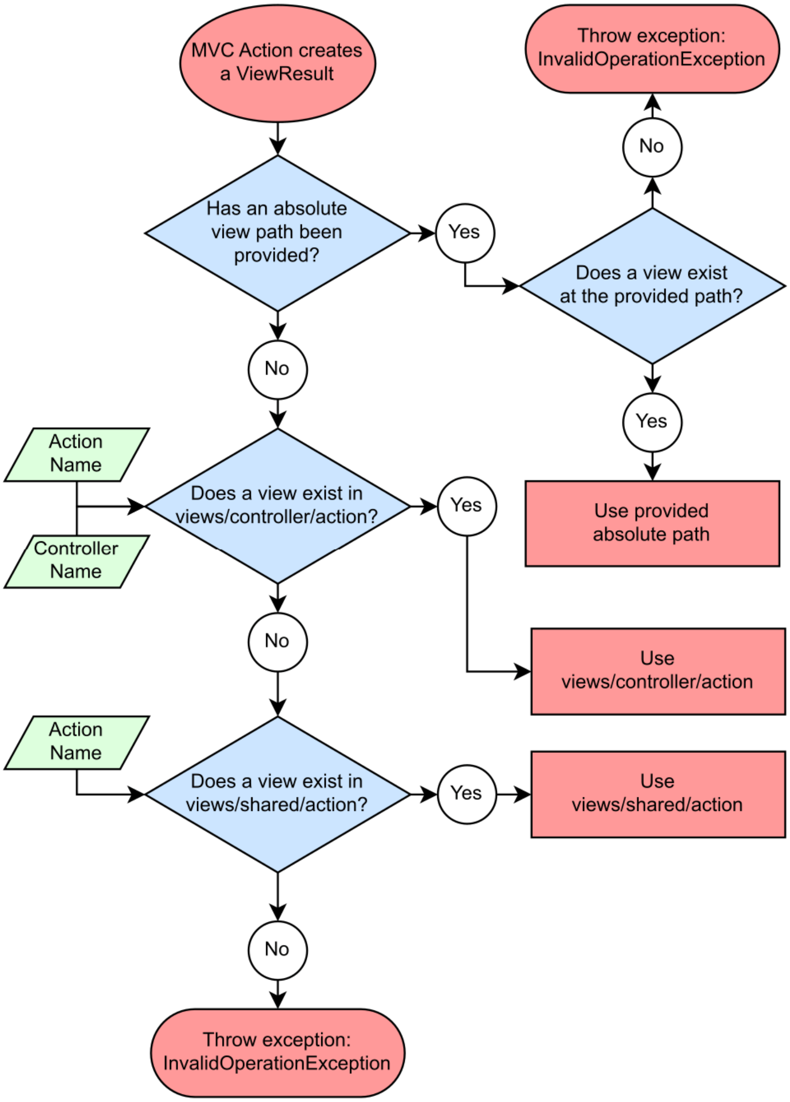

MVC Action creates a ViewResult
Has an absolute view path been provided?
Yes
Does a view exist at the provided path?
No
Throw exception: InvalidOperationException
Yes
Use provided absolute path
No
Action Name
Controller Name
Does a view exist in views/controller/action?
Yes
Use views/controller/action
No
Action Name
Does a view exist in views/shared/action?
Yes
Use views/shared/action
No
Throw exception: InvalidOperationException

Figure 7.16 A flow chart describing how the Razor templating engine locates the correct view template to execute. Avoiding the complexity of this diagram is one of the reasons I recommend using Razor Pages wherever possible!

TIP You can modify all these conventions, including the algorithm shown in figure 7.16, during initial configuration. In fact, you can replace the whole Razor templating engine if required, but that's beyond the scope of this book.

You may find it tempting to explicitly provide the name of the view file you want to render in your controller; if so, I'd encourage you to fight that urge. You'll have a much simpler time if you embrace the conventions as they are and go with the flow. That extends to anyone else who looks at your code; if you stick to the standard conventions, then there'll be a comforting familiarity when they look at your app. That can only be a good thing!

As well as providing a view template name, you can also pass an object to act as the view model for the Razor view. This object should match the type specified in the view's `@model` directive, and is accessed in exactly the same way as for Razor Pages; using the `Model` property. The listing below shows two examples of passing a view model to a view.

Listing 7.15 Returning `ViewResult` from an action method using default conventions

```
public class ToDoController : Controller
{
    public IActionResult Index()
    {
        var listViewModel = new ToDoListMdel();      #A
        return View(listViewModel);                  #B
    }

    public IActionResult View(int id)
    {
        var viewModel = new ViewToDoModel();
        return View("ViewToDo", viewModel);          #C
    }

}
```

#A Creating an instance of the view model to pass to the Razor view.
#B The view model is passed as an argument to View
#C You can provide the view template name at the same time as the view model.

Once the Razor view template has been located, the view is rendered using the Razor syntax you've seen throughout this chapter. You can use all the features you've already seen—layouts, partial views, _ViewImports, and _ViewStart for example. From the point of the view of the Razor view, there's no difference between a Razor Pages view and an MVC Razor view.

That concludes our first look at rendering HTML using the Razor templating engine. In the next chapter, you'll learn about Tag Helpers and how to use them to build HTML forms, a staple of modern web applications. Tag Helpers are one of the biggest improvements to Razor in ASP.NET Core over the previous version, so getting to grips with them will make editing your views an overall more pleasant experience!

7.6 Summary

- In the MVC design pattern, views are responsible for generating the UI for your application.
- Razor is a templating language that allows you to generate dynamic HTML using a mixture of HTML and C#.

- HTML forms are the standard approach for sending data from the browser to the server. You can use Tag Helpers to easily generate these forms.
- Razor Pages can pass strongly-typed data to a Razor view by setting public properties on the `PageModel`. To access the properties on the view model, the view should declare the model type using the `@model` directive.
- Page handlers can pass key-value pairs to the view using the `ViewData` dictionary.
- Razor expressions render C# values to the HTML output using `@` or `@()`. You don't need to include a semicolon after the statement when using Razor expressions.
- Razor code blocks, defined using `@{}`, execute C# without outputting HTML. The C# in Razor code blocks must be complete statements, so it must include semicolons.
- Loops and conditionals can be used to easily generate dynamic HTML in templates, but it's a good idea to limit the number of `if` statements in particular, to keep your views easy to read.
- If you need to render a `string` as raw HTML you can use `Html.Raw`, but do so sparingly—rendering raw user input can create a security vulnerability in your application.
- Tag Helpers allow you to bind your data model to HTML elements, making it easier to generate dynamic HTML while staying editor friendly.
- You can place HTML common to multiple views in a layout. The layout will render any content from the child view at the location `@RenderBody` is called.
- Encapsulate commonly used snippets of Razor code in a partial view. A partial view can be rendered using the `<partial />` tag.
- _ViewImports.cshtml can be used to include common directives, such as `@using` statements, in every view.
- _ViewStart.cshtml is called before the execution of each Razor Page and can be used to execute code common to all Razor Pages, such as setting a default layout page. It doesn't execute for layouts or partial views.
- _ViewImports.cshtml and _ViewStart.cshtml are hierarchical—files in the root folder execute first, followed by files in controller-specific view folders.
- Controllers can invoke a Razor view by returning a `ViewResult`. This may contain the name of the view to render and optionally a view model object to use when rendering the view. If the view name is not provided, a view is chosen using conventions.
- By convention, MVC Razor views are named the same as the action method that invokes them. They reside either in a folder with the same name as the action method's controller or in the Shared folder.

8

Building forms with Tag Helpers

This chapter covers

- Building forms easily with Tag Helpers
- Generating URLs with the Anchor Tag Helper
- Using Tag Helpers to add functionality to Razor

In chapter 7, you learned about Razor templates and how to use them to generate the views for your application. By mixing HTML and C#, you can create dynamic applications that can display different data based on the request, the logged-in user, or any other data you can access.

Displaying dynamic data is an important aspect of many web applications, but it's typically only one half of the story. As well as displaying data to the user, you often need the user to be able to submit data *back* to your application. You can use data to customize the view, or to update the application model by saving it to a database, for example. For traditional web applications, this data is usually submitted using an HTML form.

In chapter 6, you learned about model binding, which is how you *accept* the data sent by a user in a request and convert it into C# objects that you can use in your Razor Pages. You also learned about validation, and how important it is to validate the data sent in a request. You used `DataAnnotations` to define the rules associated with your models, as well as other associated metadata like the display name for a property.

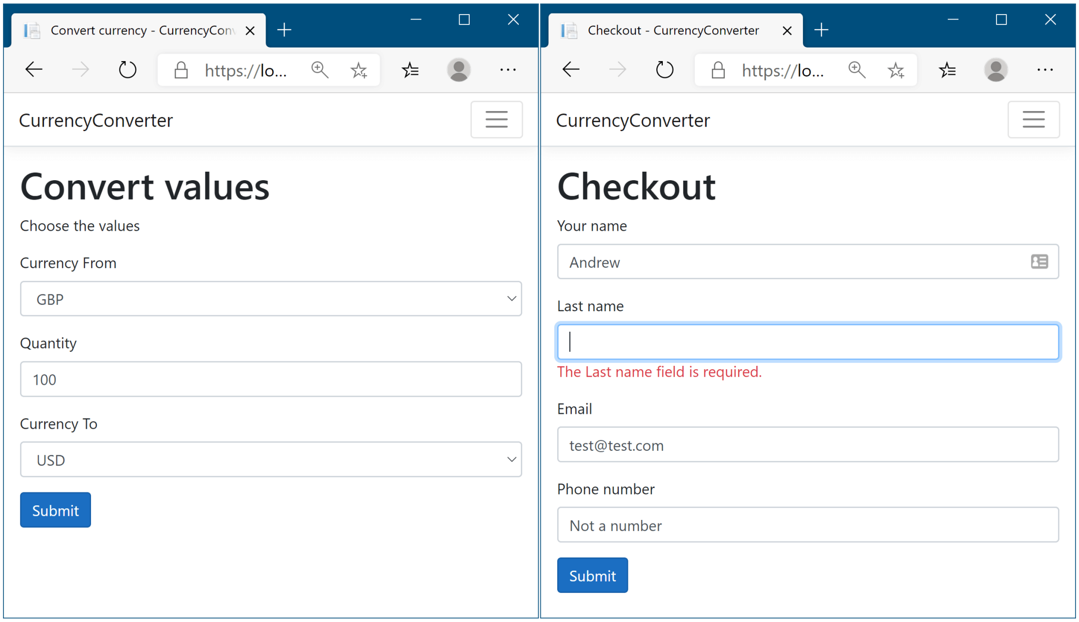
The final aspect we haven't yet looked at is how to *build* the HTML forms that users use to send this data in a request. Forms are one of the key ways users will interact with your application in the browser, so it's important they're both correctly defined for your application and also user friendly. ASP.NET Core provides a feature to achieve this, called *Tag Helpers*.

Tag Helpers are new to ASP.NET Core. They're additions to Razor syntax that you can use to customize the HTML generated in your templates. Tag Helpers can be added to an otherwise standard HTML element, such as an `<input>`, to customize its attributes based on your C# model, saving you from having to write boilerplate code. Tag Helpers can also be standalone elements and can be used to generate completely customized HTML.

> **NOTE** **Remember, Razor, and therefore Tag Helpers, are for server-side HTML rendering. You can't use Tag Helpers directly in frontend frameworks like Angular or React.**

If you've used the previous version of ASP.NET, then Tag Helpers may sound reminiscent of HTML Helpers, which could also be used to generate HTML based on your C# classes. Tag Helpers are the logical successor to HTML Helpers, as they provide a more streamlined syntax than the previous, C#-focused helpers. HTML Helpers are still available in ASP.NET Core, so if you're converting some old templates to ASP.NET Core you can still use them in your templates, but I won't be covering them in this book.

In this chapter, you'll primarily learn how to use Tag Helpers when building forms. They simplify the process of generating correct element names and IDs so that model binding can occur seamlessly when the form is sent back to your application. To put them into context, you're going to carry on building the currency converter application that you've seen in previous chapters. You'll add the ability to submit currency exchange requests to it, validate the data, and redisplay errors on the form using Tag Helpers to do the leg work for you, as shown in figure 8.1.



Convert values

Choose the values

Currency From: GBP

Quantity: 100

Currency To: USD

Checkout

Your name: Andrew

Last name:

The Last name field is required.

Email: test@test.com

Phone number: Not a number

Figure 8.1 The currency converter application forms, built using Tag Helpers. The labels, dropdowns, input elements, and validation messages are all generated using Tag Helpers.

As you develop the application, you'll meet the most common Tag Helpers you'll encounter when working with forms. You'll also see how you can use Tag Helpers to simplify other common tasks, such as generating links, conditionally displaying data in your application, and ensuring users see the latest version of an image file when they refresh their browser.

To start, I'll talk a little about why you need Tag Helpers when Razor can already generate any HTML you like by combining C# and HTML in a file.

8.1 Catering to editors with Tag Helpers

One of the common complaints about the mixture of C# and HTML in Razor templates is that you can't easily use standard HTML editing tools with them; all the `@` and `{}` symbols in the C# code tend to confuse the editors. Reading the templates can be similarly difficult for people; switching paradigms between C# and HTML can be a bit jarring sometimes.

This arguably wasn't such a problem when Visual Studio was the only supported way to build ASP.NET websites, as it could obviously understand the templates without any issues, and helpfully colorize the editor. But with ASP.NET Core going cross-platform, the desire to play nicely with other editors reared its head again.

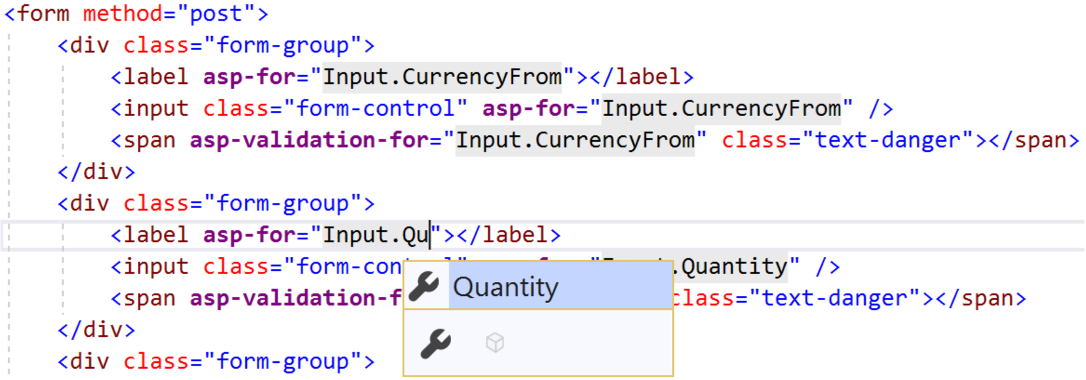
This was one of the big motivations for Tag Helpers. They integrate seamlessly into the standard HTML syntax by adding what look to be attributes, typically starting with `asp-*`. They're most often used to generate HTML forms, as shown in the following listing. This listing shows a view from the first iteration of the currency converter application, in which you choose the currencies and quantity to convert.

Listing 8.1 User registration form using Tag Helpers

```
@page                                                              #A
@model ConvertModel                                                #A
<form method="post">
    <div class="form-group">
        <label asp-for="CurrencyFrom"></label>                     #B
        <input class="form-control" asp-for="CurrencyFrom" />      #C
        <span asp-validation-for="CurrencyFrom"></span>            #D
    </div>
    <div class="form-group">
        <label asp-for="Quantity"></label>                         #B
        <input class="form-control" asp-for="Quantity" />          #C
        <span asp-validation-for="Quantity"></span>                #D
    </div>
    <div class="form-group">
        <label asp-for="CurrencyTo"></label>                       #B
        <input class="form-control" asp-for="CurrencyTo" />        #C
        <span asp-validation-for="CurrencyTo"></span>              #D
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

#A This is the view for the Razor Page Convert.cshtml. The Model type is ConvertModel.
#B asp-for on Labels generates the caption for labels based on the view model.
#C asp-for on Inputs generate the correct type, value, name, and validation attributes for the model.
#D Validation messages are written to a span using Tag Helpers.

At first glance, you might not even spot the Tag Helpers, they blend in so well with the HTML! This makes it easy to edit the files with any standard HTML text editor. But don't be concerned that you've sacrificed readability in Visual Studio—as you can see in figure 8.2, elements with Tag Helpers are clearly distinguishable from the standard HTML `<div>` element and the standard HTML `class` attribute on the `<input>` element. The C# properties of the view model being referenced (`CurrencyFrom`, in this case) are also still shaded, as with other C# code in Razor files. And of course, you get IntelliSense, as you'd expect.[31]



```
<form method="post">
    <div class="form-group">
        <label asp-for="Input.CurrencyFrom"></label>
        <input class="form-control" asp-for="Input.CurrencyFrom" />
        <span asp-validation-for="Input.CurrencyFrom" class="text-danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="Input.Qu"></label>
        <input class="form-con                     .Quantity" />
        <span asp-validation-f                     class="text-danger"></span>
    </div>
    <div class="form-group">
```

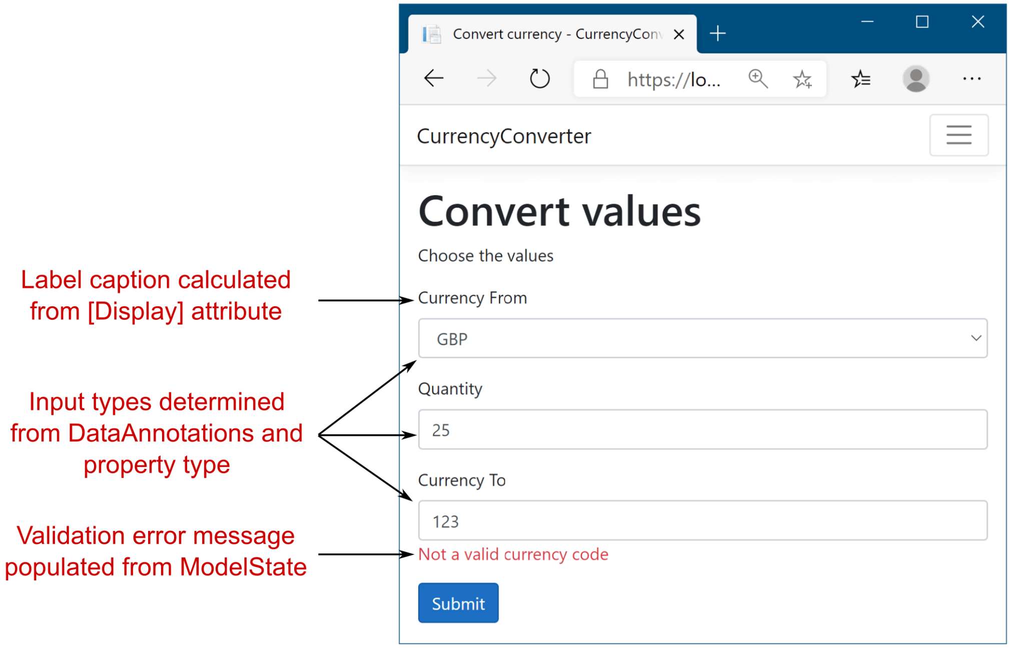
Figure 8.2 In Visual Studio, Tag Helpers are distinguishable from normal elements by being bold and a different color, C# is shaded, and IntelliSense is available.

Tag Helpers are extra attributes on standard HTML elements (or new elements entirely) that work by modifying the HTML element they're attached to. They let you easily integrate your server-side values, such as those exposed on your `PageModel`, with the generated HTML.

Notice that listing 8.1 didn't specify the captions to display in the labels. Instead, you declaratively used `asp-for="CurrencyFrom"` to say, "for this `<label>`, use the `CurrencyFrom` property to work out what caption to use." Similarly, for the `<input>` elements, Tag Helpers are used to

- Automatically populate the value from the `PageModel` property
- Choose the correct `id` and `name`, so that when the form is `POST`ed back to the Razor Page, the property will be model bound correctly.
- Choose the correct input type to display (for example, a `number` input for the `Quantity` property)
- Display any validation errors, as shown in figure 8.3[32]

[31] Other editors like Visual Studio Code, JetBrains Rider, and Visual Studio for Mac also include syntax highlighting and IntelliSense support.
[32] To learn more about the internals of Tag Helpers, read the documentation at http://mng.bz/Idb0.



Convert values

Choose the values

Currency From

GBP

Quantity

25

Currency To

123

Not a valid currency code

Convert currency - CurrencyConv
https://lo...
CurrencyConverter
Submit
Label caption calculated from [Display] attribute
Input types determined from DataAnnotations and property type
Validation error message populated from ModelState

Figure 8.3 Tag Helpers hook into the metadata provided by `DataAnnotation`s, as well as the property types themselves. The Validation Tag Helper can even populate error messages based on the `ModelState`, as you saw in the last chapter on validation.

Tag Helpers can perform a variety of functions by modifying the HTML elements they're applied to. This chapter introduces a number of the common Tag Helpers and how to use them, but it's not an exhaustive list. I don't cover all of the helpers that come out of the box in ASP.NET Core (there are more coming with every release!), and you can easily create your own, as you'll see in chapter 19. Alternatively, you could use those published by others on NuGet or GitHub[33]. As with all of ASP.NET Core, Microsoft is developing Tag Helpers in the open on GitHub, so you can always take a look at the source code to see how they're implemented.

WebForms flashbacks

For those who used ASP.NET back in the day of WebForms, before the advent of the MVC pattern for web development, Tag Helpers may be triggering bad memories. Although the `asp-` prefix is somewhat reminiscent of ASP.NET Web Server control definitions, never fear—the two are different beasts.

[33] A good example is Damian Edwards' (of the ASP.NET Core team) Tag Helper pack https://github.com/DamianEdwards/TagHelperPack.

Web Server controls were directly added to a page's backing C# class, and had a broad scope that could modify seemingly unrelated parts of the page. Coupled with that, they had a complex lifecycle that was hard to understand and debug when things weren't working. The perils of trying to work with that level of complexity haven't been forgotten, and Tag Helpers aren't the same.

Tag Helpers don't have a lifecycle—they participate in the rendering of the element to which they're attached, and that's it. They can modify the HTML element they're attached to, but they can't modify anything else on your page, making them conceptually much simpler. An additional capability they bring is the ability to have multiple Tag Helpers acting on a single element—something Web Server controls couldn't easily achieve.

Overall, if you're writing Razor templates, you'll have a much more enjoyable experience if you embrace Tag Helpers as integral to its syntax. They bring a lot of benefits without obvious downsides, and your cross-platform-editor friends will thank you!

8.2 Creating forms using Tag Helpers

In this section you'll learn how to use some of the most useful Tag Helpers: Tag helpers that work with forms. You'll learn how to use them to generate HTML markup based on properties of your `PageModel`, creating the correct `id` and `name` attributes and setting the `value` of the element to the model property's value (among other things). This capability significantly reduces the amount of markup you need to write manually.

Imagine you're building the checkout page for the currency converter application, and you need to capture the user's details on the checkout page. In chapter 6, you built a `UserBindingModel` model (shown in listing 8.2), added `DataAnnotation` attributes for validation, and saw how to model bind it in a `POST` to a Razor Page. In this chapter, you'll see how to create the view for it, by exposing the `UserBindingModel` as a property on your `PageModel`.

> **WARNING** With Razor Pages, you often expose the same object in your view that you use for model binding. When you do this, you must be careful to not include sensitive values (that shouldn't be edited) in the binding model, to avoid mass-assignment attacks on your app.[34]

Listing 8.2 `UserBindingModel` for creating a user on a checkout page

```
public class UserBindingModel
{
    [Required]
    [StringLength(100, ErrorMessage = "Maximum length is {1}")]
    [Display(Name = "Your name")]
    public string FirstName { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "Maximum length is {1}")]
```

[34] You can read more about over posting attacks at https://andrewlock.net/preventing-mass-assignment-or-over-posting-with-razor-pages-in-asp-net-core/.

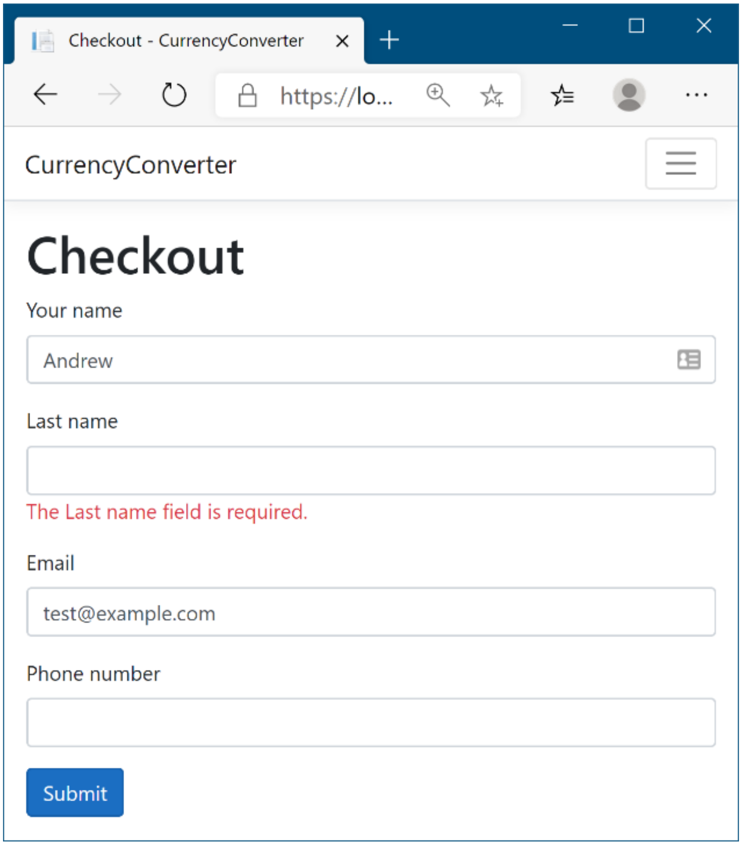
```
    [Display(Name = "Last name")]
    public string LastName { get; set; }

    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Phone(ErrorMessage = "Not a valid phone number.")]
    [Display(Name = "Phone number")]
    public string PhoneNumber { get; set; }
}
```

The `UserBindingModel` is decorated with a number of `DataAnnotations` attributes. In chapter 6, you saw that these attributes are used during model validation when the model is bound to a request, before the page handler is executed. These attributes are *also* used by the Razor templating language to provide the metadata required to generate the correct HTML when you use Tag Helpers.

You can use the pattern I described in chapter 6, exposing a `UserBindindModel` as an `Input` property of your `PageModel` to use the model for both model binding and in your Razor view:

```
public class CheckoutModel: PageModel
{
    [BindProperty]
    public UserBindingModel Input { get; set; }
}
```

With the help of the `UserBindingModel` property, Tag Helpers, and a little HTML, you can create a Razor view that lets the user enter their details, as shown in figure 8.4.



Checkout

Your name

Andrew

Last name

The Last name field is required.

Email

test@example.com

Phone number

Figure 8.4 The checkout page for an application. The HTML is generated based on a `UserBindingModel`, using Tag Helpers to render the required element values, input types, and validation messages.

The Razor template to generate this page is shown in listing 8.3. This code uses a variety of tag helpers, including

- A Form Tag Helper on the `<form>` element
- Label Tag Helpers on the `<label>`
- Input Tag Helpers on the `<input>`
- Validation Message Tag Helpers on `<span>` validation elements for each property in the `UserBindingModel`

Listing 8.3 Razor template for binding to `UserBindingModel` on the checkout page

```
@page
@model CheckoutModel                                          #A
@{
    ViewData["Title"] = "Checkout";
}
<h1>@ViewData["Title"]</h1>
<form asp-page="Checkout">                                          #B
    <div class="form-group">
        <label asp-for="Input.FirstName"></label>                   #C
        <input class="form-control" asp-for="Input.FirstName" />
        <span asp-validation-for="Input.FirstName"></span>
    </div>
    <div class="form-group">
        <label asp-for="Input.LastName"></label>
        <input class="form-control" asp-for="Input.LastName" />
        <span asp-validation-for="Input.LastName"></span>
    </div>
    <div class="form-group">
        <label asp-for="Input.Email"></label>
        <input class="form-control" asp-for="Input.Email" />           #D
        <span asp-validation-for="Input.Email"></span>
    </div>
    <div class="form-group">
        <label asp-for="Input.PhoneNumber"></label>
        <input class="form-control" asp-for="Input.PhoneNumber" />
        <span asp-validation-for="Input.PhoneNumber"></span>           #E
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

#A The CheckoutModel is the PageModel, which exposes a UserBindingModel on the Input property
#B Form Tag Helpers use routing to determine the URL the form will be posted to.
#C The Label Tag Helper uses DataAnnotations on a property to determine the caption to display.
#D The Input Tag Helper uses DataAnnotations to determine the type of input to generate.
#E The Validation Tag Helper displays error messages associated with the given property.

You can see the HTML markup that this template produces in listing 8.4. This Razor markup and the resulting HTML produces the results you saw in figure 8.4. You can see that each of the HTML elements with a Tag Helper has been customized in the output: the `<form>` element has an `action` attribute, the `<input>` elements have an `id` and `name` based on the name of the referenced property, and both the `<input>` and `<span>` have `data-*` elements for validation.

Listing 8.4 HTML generated by the Razor template on the checkout page

```
<form action="/Checkout" method="post">
  <div class="form-group">
    <label for="Input_FirstName">Your name</label>
    <input class="form-control" type="text"
      data-val="true" data-val-length="Maximum length is 100"
      id="Input_FirstName" data-val-length-max="100"
      data-val-required="The Your name field is required."
      maxlength="100" name="Input.FirstName" value="" />
    <span data-valmsg-for="Input.FirstName"
      class="field-validation-valid" data-valmsg-replace="true"></span>
  </div>
  <div class="form-group">
    <label for="Input_LastName">Your name</label>
    <input class="form-control" type="text"
      data-val="true" data-val-length="Maximum length is 100"
      id="Input_LastName" data-val-length-max="100"
      data-val-required="The Your name field is required."
      maxlength="100" name="Input.LastName" value="" />
    <span data-valmsg-for="Input.LastName"
      class="field-validation-valid" data-valmsg-replace="true"></span>
  </div>
  <div class="form-group">
    <label for="Input_Email">Email</label>
    <input class="form-control" type="email" data-val="true"
      data-val-email="The Email field is not a valid e-mail address."
      data-val-required="The Email field is required."
      id="Input_Email" name="Input.Email" value="" />
    <span class="text-danger field-validation-valid"
      data-valmsg-for="Input.Email" data-valmsg-replace="true"></span>
    </div>
  <div class="form-group">
    <label for="Input_PhoneNumber">Phone number</label>
    <input class="form-control" type="tel" data-val="true"
      data-val-phone="Not a valid phone number." id="Input_PhoneNumber"
      name="Input.PhoneNumber" value="" />
    <span data-valmsg-for="Input.PhoneNumber"
      class="text-danger field-validation-valid"
      data-valmsg-replace="true"></span>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
  <input name="__RequestVerificationToken" type="hidden"
    value="CfDJ8PkYhAINFx1JmYUVIDWbpPyy_TRUNCATED" />
</form>
```

Wow, that's a lot of markup! If you're new to working with HTML, this might all seem a little overwhelming, but the important thing to notice is that *you didn't have to write most of it*! The Tag Helpers took care of most of the plumbing for you. That's basically Tag Helpers in a nutshell; they simplify the fiddly mechanics of building HTML forms, leaving you to concentrate on the overall design of your application instead of writing boilerplate markup.

> **NOTE** **If you're using Razor to build your views, Tag Helpers will make your life easier, but they're entirely optional. You're free to write raw HTML without them, or to use the legacy HTML Helpers.**

Tag Helpers simplify and abstract the process of HTML generation, but they generally try to do so without getting in your way. If you need the final generated HTML to have a particular attribute, then you can add it to your markup. You can see that in the previous listings where `class` attributes are defined on `<input>` elements, such as `<input class="form-control" asp-for="Input.FirstName" />`, they pass untouched through from Razor to the HTML output.

> **TIP** **This is different from the way HTML Helpers worked in the previous version of ASP.NET; HTML helpers often require jumping through hoops to set attributes in the generated markup.**

Even better than this, you can also set attributes that are normally generated by a Tag Helper, like the `type` attribute on an `<input>` element. For example, if the `FavoriteColor` property on your PageModel was a string, then by default, Tag Helpers would generate an <input> element with type="text". Updating your markup to use the HTML5 color picker type is trivial; set the type explicitly in your Razor view:

```
<input type="color" asp-for="FavoriteColor" />
```

TIP **HTML5 adds a huge number of features, including lots of form elements that you may not have come across before, such as range inputs and color pickers. We're not going to cover them in this book, but you can read about them on the Mozilla Developer Network website at http://mng.bz/qOc1.**

In this section, you'll build the currency calculator Razor templates from scratch, adding Tag Helpers as you find you need them. You'll probably find you use most of the common form Tag Helpers in every application you build, even if it's on a simple login page.

8.2.1 The Form Tag Helper

The first thing you need to start building your HTML form is, unsurprisingly, the <form> element. In the previous example, the <form> element was augmented with a Tag Helper attribute: asp-page:

```
<form asp-page="Checkout">
```

This results in the addition of action and method attributes to the final HTML, indicating the URL that the form should be sent to when submitted and the HTTP verb to use:

```
<form action="/Checkout" method="post">
```

Setting the asp-page attribute allows you to specify a different Razor Page in your application that the form will be posted to when it's submitted. If you omit the asp-page attribute, the form will post back to the same URL address it was served from. This is *very* common with Razor Pages. You normally handle the result of a form post in the same Razor Page that is used to display it.

WARNING **If you omit the asp-page attribute you must manually add the method="post" attribute. It's important to add this attribute, so the form is sent using the POST verb, instead of the default GET verb. Using GET for forms can be a security risk.**

The asp-page attribute is added by a FormTagHelper. This Tag Helper uses the value provide to generate a URL for the action attribute using the URL generation features of routing that I described at the end of chapter 5.

NOTE **Tag Helpers can make multiple attributes available on an element. Think of them like properties on a Tag Helper configuration object. Adding a single asp- attribute activates the Tag Helper on the element. Adding additional attributes lets you override further default values of its implementation.**

The Form Tag Helper makes several other attributes available on the `<form>` element that you can use to customize the generated URL. Hopefully you'll remember from chapter 5 that you can set route values when generating URLs. For example, if you have a Razor Page called Product.cshtml which uses the directive

```
@page "{id}"
```

Then the full route template for the page would be `"Product/{id}"`. To correctly generate the URL for this page, you must provide the `{id}` route value. How can you set that value using the Form Tag Helper?

The Form Tag Helper defines an `asp-route-*` wildcard attribute that you can use to set arbitrary route parameters. Set the `*` in the attribute to the route parameter name. For example, to set the `id` route parameter, you'd set the `asp-route-id` value (shown with a fixed value of `5` in the example below, but more commonly would be dynamic):

```
<form asp-page="Product" asp-route-id="5">
```

Based on the route template of the Product.cshtml Razor Page, this would generate the following markup:

```
<form action="/Product/5" method="post">
```

You can add as many `asp-route-*` attributes as necessary to your `<form>` to generate the correct `action` URL. You can also set the Razor Page handler to use using the `asp-page-handler` attribute. This ensures the form `POST` will be handled by the handler you specify.

NOTE **The Form Tag Helper has many additional attributes, such as `asp-action` and `asp-controller`, that you generally won't need to use with Razor Pages. Those are only useful if you're using MVC controllers with views. In particular, look out for the `asp-route` attribute—this is *not* the same as the `asp-route-*` attribute. The former is used to specify a *named* route (not used with Razor Pages), and the latter is used to specify the route *values* to use during URL generation.**

Just as for all other Razor constructs, you can use C# values from your `PageModel` (or C# in general) in Tag Helpers. For example, if the `ProductId` property of your `PageModel` contains the value required for the `{id}` route value, you could use

```
<form asp-page="Product" asp-route-id="@Model.ProductId">
```

The main job of the Form Tag Helper is to generate the `action` attribute, but it performs one additional, important function: generating a hidden `<input>` field needed to prevent *cross-site request forgery* (CSRF) attacks.

DEFINITION ***Cross-site request forgery* (CSRF) attacks are a website exploit that can be used to execute actions on your website by an unrelated malicious website. You'll learn about them in detail in chapter 18.**

You can see the generated hidden `<input>` at the bottom of the `<form>` in listing 8.4; it's named `__RequestVerificationToken` and contains a seemingly random string of characters. This field won't protect you on its own, but I'll describe in chapter 18 how it's used to protect your website. The Form Tag Helper generates it by default, so generally speaking you won't need to worry about it, but if you need to disable it, you can do so by adding `asp-antiforgery="false"` to your `<form>` element.

The Form Tag Helper is obviously useful for generating the `action` URL, but it's time to move on to more interesting elements, those that you can see in your browser!

8.2.2 The Label Tag Helper

Every `<input>` field in your currency converter application needs to have an associated label so the user knows what the `<input>` is for. You could easily create those yourself, manually typing the name of the field and setting the `for` attribute as appropriate, but luckily there's a Tag Helper to do that for you.

The Label Tag Helper is used to generate the caption (the visible text) and the `for` attribute for a `<label>` element, based on the properties in the `PageModel`. It's used by providing the name of the property in the `asp-for` attribute:

```
<label asp-for="FirstName"></label>
```

The Label Tag Helper uses the `[Display] DataAnnotations` attribute that you saw in chapter 6 to determine the appropriate value to display. If the property you're generating a label for doesn't have a `[Display]` attribute, the Label Tag Helper will use the name of the property instead. So, for the model

```
public class UserModel
{
    [Display(Name = "Your name")]
    public string FirstName { get; set; }
    public string Email { get; set; }
}
```

in which the `FirstName` property has a `[Display]` attribute, but the `Email` property doesn't; the following Razor

```
<label asp-for="FirstName"></label>
<label asp-for="Email"></label>
```

would generate the HTML

```
<label for="FirstName">Your Name</label>
<label for="Email">Email</label>
```

The caption text inside the `<label>` element uses the value set in the `[Display]` attribute, or the property name in the case of the `Email` property. Also note that the `for` attribute has been generated with the name of the property. This is a key bonus of using Tag Helpers—it hooks in with the element IDs generated by other Tag Helpers, as you'll see shortly.

> **NOTE** The `for` attribute is important for accessibility. It specifies the ID of the element to which the label refers. This is important for users that are using a screen reader, for example, as they can tell what property a form field relates to.

As well as properties on the `PageModel`, you can also reference sub-properties on child objects. For example, as I described in chapter 6, it's common to create a nested class in a Razor Page, expose that as a property, and decorate it with the `[BindProperty]` attribute:

```
public class CheckoutModel: PageModel
{
    [BindProperty]
    public UserBindingModel Input { get; set; }
}
```

You can reference the `FirstName` property of the `UserBindingModel` by "dotting" into the property as you would in any other C# code. Listing 8.3 shows more examples of this.

```
<label asp-for="Input.FirstName"></label>
<label asp-for="Input.Email"></label>
```

As is typical with Tag Helpers, the Label Tag Helper won't override values that you set yourself. If, for example, you don't want to use the caption generated by the helper, you could insert your own manually. The following code

```
<label asp-for="Email">Please enter your Email</label>
```

would generate the HTML

```
<label for="Email">Please enter your Email</label>
```

As ever, you'll generally have an easier time with maintenance if you stick to the standard conventions and don't override values like this, but the option is there. Right, next up is a biggie: the Input and Textarea Tag Helpers.

8.2.3 The Input and Textarea Tag Helpers

Now you're getting into the meat of your form—the `<input>` elements that handle user input. Given that there's such a wide array of possible input types, there's a variety of different ways they can be displayed in the browser. For example, Boolean values are typically represented by a `checkbox` type `<input>` element, whereas integer values would use a `number` type `<input>` element, and a date would use the `date` type, shown in figure 8.5.

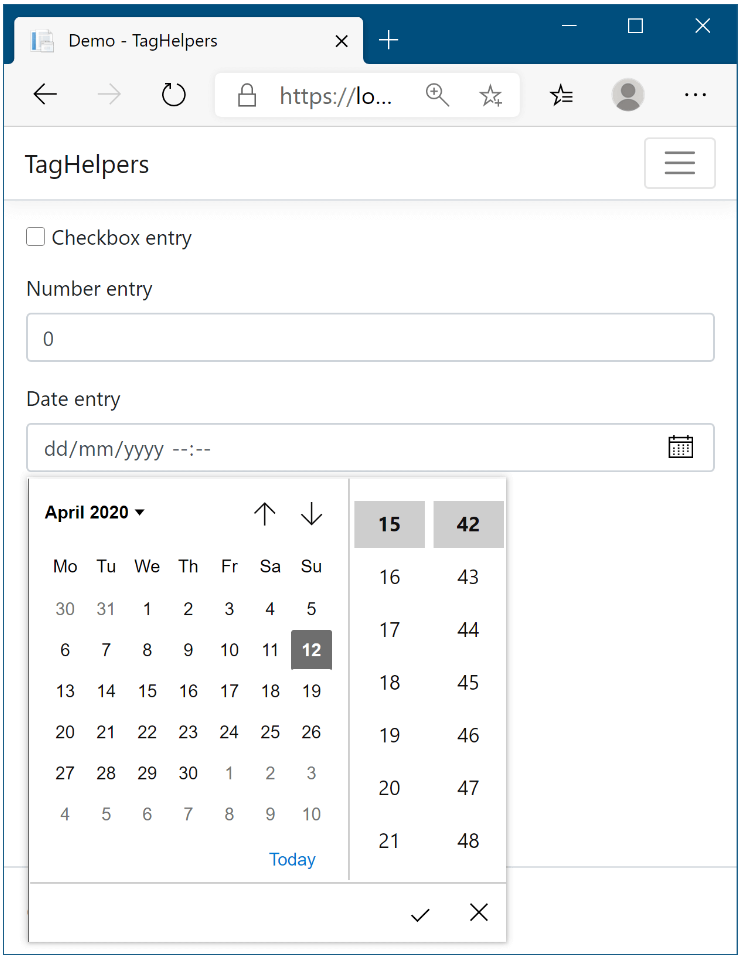


Figure 8.5 Various input element types. The exact way in which each type is displayed varies by browser.

To handle this diversity, the Input Tag Helper is one of the most powerful Tag Helpers. It uses information based on both the type of the property (`bool`, `string`, `int`, and so on) and any `DataAnnotations` attributes applied to it (`[EmailAddress]` and `[Phone]`, among others) to determine the type of the `input` element to generate. The `DataAnnotations` are also used to add `data-val-*` client-side validation attributes to the generated HTML.

Consider the `Email` property from listing 8.2 that was decorated with the `[Email-Address]` attribute. Adding an `<input>` is as simple as using the `asp-for` attribute:

```
<input asp-for="Input.Email" />
```

The property is a string, so ordinarily, the Input Tag Helper would generate an <input> with type="text". But the addition of the [EmailAddress] attribute provides additional metadata about the property. Consequently, the Tag Helper generates an HTML5 <input> with type="email":

```
<input type="email" id="Input_Email" name="Input.Email"
    value="test@example.com" data-val="true"
    data-val-email="The Email Address field is not a valid e-mail address."
    data-val-required="The Email Address field is required."
    />
```

You can take a whole host of things away from this example. First, the id and name attributes of the HTML element have been generated from the name of the property. The value of the id attribute matches the value generated by the Label Tag Helper in its for attribute, Input_Email. The value of the name attribute preserves the "dot" notation, Input.Email, so that model binding works correctly when the field is POSTed to the Razor Page.

Also, the initial value of the field has been set to the value currently stored in the property ("test@example.com", in this case). The type of the element has also been set to the HTML5 email type, instead of using the default text type.

Perhaps the most striking addition is the swath of data-val-* attributes. These can be used by client-side JavaScript libraries such as jQuery to provide client-side validation of your DataAnnotations constraints. Client-side validation provides instant feedback to users when the values they enter are invalid, providing a smoother user experience than can be achieved with server-side validation alone, as I described in chapter 6.

Client-side validation

In order to enable client-side validation in your application, you need to add some jQuery libraries to your HTML pages. In particular, you need to include the jQuery, jQuery-validation, and jQuery-validation-unobtrusive JavaScript libraries. You can do this in a number of ways, but the simplest is to include the script files at the bottom of your view using

```
<script src="~/lib/jquery-validation/dist/jquery.validate.min.js"></script>
<script src="~/lib/jquery-validation-
unobtrusive/jquery.validate.unobtrusive.min.js"></script>
```

The default templates include these scripts for you, in a handy partial template that you can add to your page in a Scripts **section. If you're using the default layout and need to add client-side validation to your view, add the following section somewhere on your view:**

```
@section Scripts{
    @Html.Partial("_ValidationScriptsPartial")
}
```

This partial view references files in your wwwroot folder. The default _layout template includes jQuery itself, as that's required by the front-end component library Bootstrap.[35]

The Input Tag Helper tries to pick the most appropriate template for a given property based on `DataAnnotation` attributes or the type of the property. Whether this generates the exact `<input>` type you need may depend, to an extent, on your application. As always, you can override the generated `type` by adding your own `type` attribute to the Razor. Table 8.1 shows how some of the common data types are mapped to `<input>` types, and how the data types themselves can be specified.

Table 8.1 Common data types, how to specify them, and the input element type they map to

Data type	How it's specified	Input element type
`byte, int, short, long, uint`	**Property type**	`number`
`decimal, double, float`	**Property type**	`text`
`string`	**Property type,** `[DataType(DataType.Text)]` **attribute**	`text`
`HiddenInput`	`[HiddenInput]` **attribute**	`hidden`
`Password`	`[Password]` **attribute**	`password`
`Phone`	`[Phone]` **attribute**	`tel`
`EmailAddress`	`[EmailAddress]` **attribute**	`email`
`Url`	`[Url]` **attribute**	`url`
`Date`	`DateTime` **property type,** `[DataType(DataType.Date)]` **attribute**	`date`

The Input Tag Helper has one additional attribute that can be used to customize the way data is displayed: `asp-format`. HTML forms are entirely string-based, so when the `value` of an `<input>` is set, the Input Tag Helper must take the value stored in the property and convert it to a `string`. Under the covers, this performs a `string.Format()` on the property's value, passing in the format string.

[35] You can also load these files from a Content Delivery Network. If you wish to take this approach, you should consider scenarios where the CDN is unavailable or compromised, as I discuss in this post: https://andrewlock.net/using-jquery-and-bootstrap-from-a-cdn-with-fallback-scripts-in-asp-net-core-3/.

The Input Tag Helper uses a default format string for each different data type, but with the asp-format attribute, you can set the specific format string to use. For example, you could ensure a decimal property, Dec, is formatted to three decimal places with the following code:

```
<input asp-for="Dec" asp-format="{0:0.000}" />
```

If the Dec property had a value of 1.2, this would generate HTML similar to

```
<input type="text" id="Dec" name="Dec" value="1.200">
```

> **NOTE** You may be surprised that decimal and double types are rendered as text fields and not as number fields. This is due to several technical reasons, predominantly related to the way some cultures render numbers with commas and spaces. Rendering as text avoids errors that would only appear in certain browser-culture combinations.

In addition to the Input Tag Helper, ASP.NET Core provides the Textarea Tag Helper. This works in a similar way, using the asp-for attribute, but is attached to a <textarea> element instead:

```
<textarea asp-for="BigtextValue"></textarea>
```

This generates HTML similar to the following. Note that the property value is rendered inside the Tag, and data-val-* validation elements are attached as usual:

```
<textarea data-val="true" id="BigtextValue" name="BigtextValue"
    data-val-length="Maximum length 200." data-val-length-max="200"
    data-val-required="The Multiline field is required." >This is some text,
I'm going to display it
in a text area</textarea>
```

Hopefully, this section has hammered home how much typing Tag Helpers can cut down on, especially when using them in conjunction with DataAnnotations for generating validation attributes. But this is more than reducing the number of keystrokes required; Tag Helpers ensure that the markup generated is *correct*, and has the correct name, id, and format to automatically bind your binding models when they're sent to the server.

With <form>, <label>, and <input> under your belt, you're able to build most of your currency converter forms. Before we look at displaying validation messages, there's one more element to look at: the <select>, or drop-down, input.

8.2.4 The Select Tag Helper

As well as <input> fields, a common element you'll see on web forms is the <select> element, or dropdowns and list boxes. Your currency converter application, for example, could use a <select> element to let you pick which currency to convert from a list.

By default, this element shows a list of items and lets you select one, but there are several variations, as shown in figure 8.6. As well as the normal drop-down box, you could show a list box, add multiselection, or display your list items in groups.

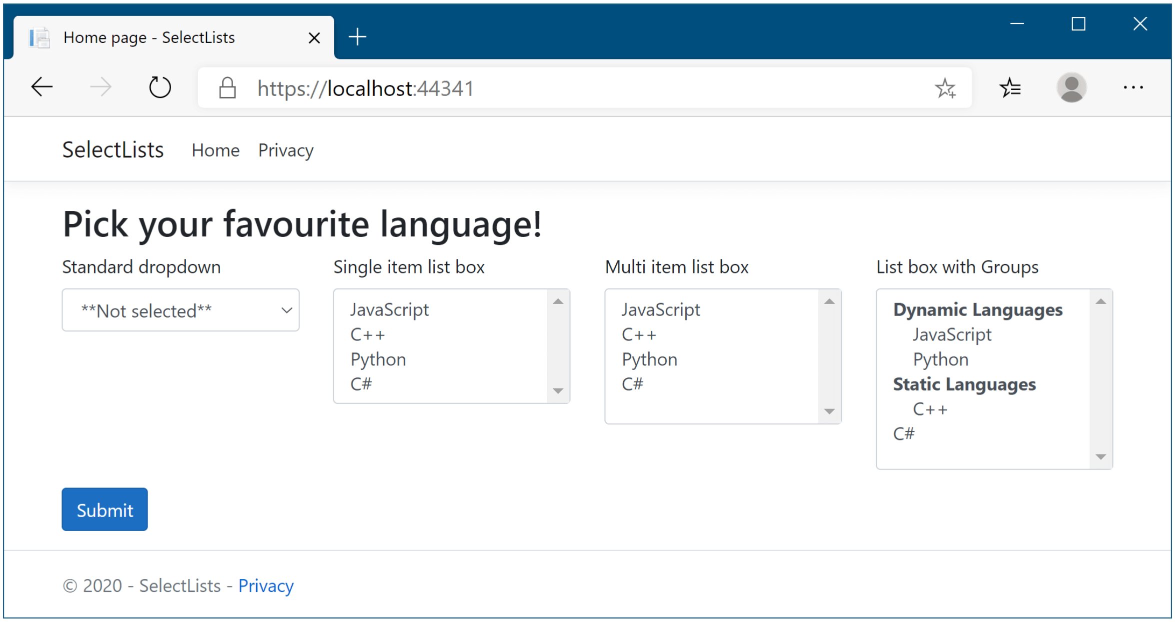


Figure 8.6 Some of the many ways to display `<select>` elements using the Select Tag Helper.

To use `<select>` elements in your Razor code, you'll need to include two properties in your `PageModel`: one property for the list of options to display and one to hold the value (or values) selected. For example, listing 8.5 shows the properties on the `PageModel` used to create the three left-most select lists you saw in figure 8.6. Displaying groups requires a slightly different setup, as you'll see shortly.

Listing 8.5 View model for displaying select element dropdowns and list boxes

```
public class SelectListsModel: PageModel
{
    [BindProperty]                                                #A
    public class InputModel Input { get; set; }                   #A

    public IEnumerable<SelectListItem> Items { get; set; }        #B
        = new List<SelectListItem>                                #B
    {                                                             #B
        new SelectListItem{Value= "csharp", Text="C#"},           #B
        new SelectListItem{Value= "python", Text= "Python"},      #B
        new SelectListItem{Value= "cpp", Text="C++"},             #B
        new SelectListItem{Value= "java", Text="Java"},           #B
        new SelectListItem{Value= "js", Text="JavaScript"},       #B
        new SelectListItem{Value= "ruby", Text="Ruby"},           #B
    };                                                            #B


    public class InputModel
    {
        public string SelectedValue1 { get; set; }              #C
        public string SelectedValue2 { get; set; }              #C
        public IEnumerable<string> MultiValues { get; set; }    #D
    }
}
```

#A The InputModel for binding the user's selections to the select boxes
#B The list of items to display in the select boxes
#C These properties will hold the values selected by the single-selection select boxes.
#D To create a multiselect list box, use an IEnumerable<>.

This listing demonstrates a number of aspects of working with `<select>` lists:

- `SelectedValue1`/`SelectedValue2`—Used to hold the values selected by the user. They're model-bound to the value selected from the drop-down/listbox and used to pre-select the correct item when rendering the form.
- `MultiValues`—Used to hold the selected values for a multiselect list. It's an `IEnumerable`, so it can hold more than one selection per `<select>` element.
- `Items`—Provides the list of options to display in the `<select>` elements. Note that the element type must be `SelectListItem`, which exposes the `Value` and `Text` properties, to work with the Select Tag Helper. This isn't part of the `InputModel`, as we don't want to model-bind these items to the request—they would normally be loaded directly from the application-model or hard-coded.

NOTE The Select Tag Helper only works with `SelectListItem` elements. That means you'll normally have to convert from an application-specific list set of items (for example, a `List<string>` or `List<MyClass>`) to the UI-centric `List<SelectListItem>`.

The Select Tag Helper exposes the `asp-for` and `asp-items` attributes that you can add to `<select>` elements. As for the Input Tag Helper, the `asp-for` attribute specifies the property in your `PageModel` to bind to. The `asp-items` attribute is provided for the `IEnumerable<SelectListItem>` to display the available `<option>` elements.

TIP It's common to want to display a list of `enum` options in a `<select>` list. This is so common that ASP.NET Core ships with a helper for generating a `SelectListItem` for any `enum`. If you have an `enum` of the `TEnum` type, you can generate the available options in your View using `asp-items="Html.GetEnumSelectList<TEnum>()"`.

The following listing shows how to display a drop-down list, a single-selection list box, and a multiselection list box. It uses the `PageModel` from the previous listing, binding each `<select>` list to a different property, but reusing the same `Items` list for all of them.

Listing 8.6 Razor template to display a select element in three different ways

```
@page
@model SelectListsModel
```

```
<select asp-for="Input.SelectedValue1"                  #A
    asp-items="Model.Items"></select>                   #A
<select asp-for="Input.SelectedValue2"                  #B
    asp-items="Model.Items" size="4"></select>          #B
<select asp-for="Input.MultiValues"                     #C
    asp-items="Model.Items"></select>                   #C
```

#A Creates a standard drop-down select list by binding to a standard property in asp-for
#B Creates a single-select list box of height 4 by providing the standard HTML size attribute
#C Creates a multiselect list box by binding to an IEnumerable property in asp-for

Hopefully, you can see that the Razor for generating a drop-down `<select>` list is almost identical to the Razor for generating a multiselect `<select>` list. The Select Tag Helper takes care of adding the `multiple` HTML attribute to the generated output if the property it's binding to is an `IEnumerable`.

> **WARNING The `asp-for` attribute *must not* include the `Model.` prefix. The `asp-items` attribute, on the other hand, *must* include it if referencing a property on the `PageModel`. The `asp-items` attribute can also reference other C# items, such as objects stored in `ViewData`, but using a `PageModel` property is the best approach.**

You've seen how to bind three different types of select list so far, but the one I haven't yet covered from figure 8.6 is how to display groups in your list boxes using `<optgroup>` elements. Luckily, nothing needs to change in your Razor code, you just have to update how you define your `SelectListItem`s.

The `SelectListItem` object defines a `Group` property that specifies the `SelectListGroup` the item belongs to. The following listing shows how you could create two groups and assign each list item to either a "dynamic" or "static" group, using a `PageModel` similar to that shown in listing 8.5. The final list item, `C#`, isn't assigned to a group, so it will be displayed as normal, without an `<optgroup>`.

Listing 8.7 Adding `Groups` to `SelectListItems` to create `optgroup` elements

```
public class SelectListsModel: PageModel
{
    [BindProperty]
    public IEnumerable<string> SelectedValues { get; set; }
    public IEnumerable<SelectListItem> Items { get; set; }

    public SelectListsModel()                                   #A
    {
        var dynamic = new SelectListGroup { Name = "Dynamic" };     #B
        var stat = new SelectListGroup { Name = "Static" };         #B
        Items = new List<SelectListItem>
        {
            new SelectListItem {
                Value= "js",
                Text="Javascript",
                Group = dynamic                                 #C
            },
            new SelectListItem {
                Value= "cpp",
                Text="C++",
                Group = stat                                          #C
            },
            new SelectListItem {
                Value= "python",
                Text="Python",
                Group = dynamic                                      #C
            },
            new SelectListItem {                                     #D
                Value= "csharp",                                     #D
                Text="C#",                                           #D
            }
        };
    }
}
```

#A Initializes the list items in the constructor
#B Creates single instance of each group to pass to SelectListItems
#C Sets the appropriate group for each SelectListItem
#D If a SelectListItem doesn’t have a Group, it won’t be added to an <optgroup>.

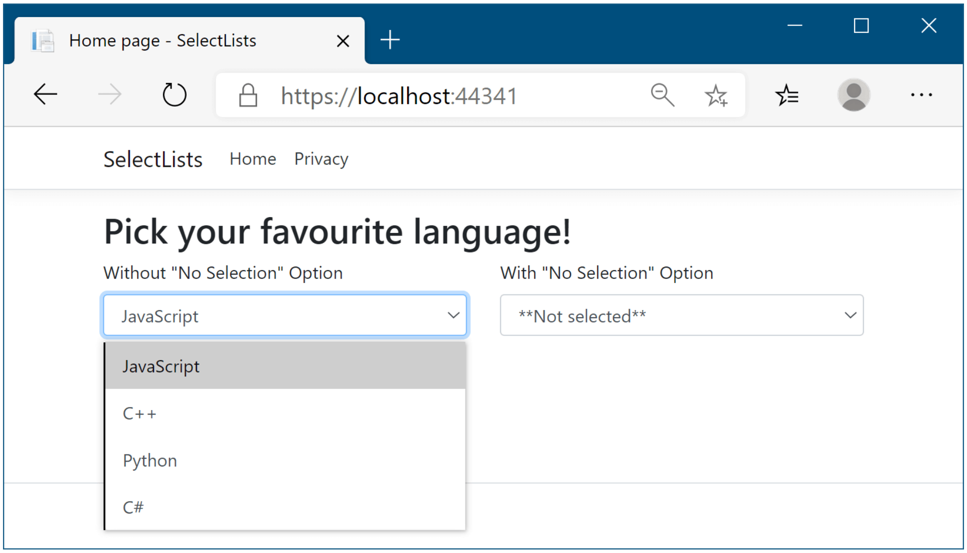
With this in place, the Select Tag Helper will generate `<optgroup>` elements as necessary when rendering the Razor to HTML. The Razor template:

```
@page
@model SelectListsModel
<select asp-for="SelectedValues" asp-items="Model.Items"></select>
```

would be rendered to HTML as:

```
<select id="SelectedValues" name="SelectedValues" multiple="multiple">
    <optgroup label="Dynamic">
        <option value="js">JavaScript</option>
        <option value="python">Python</option>
    </optgroup>
    <optgroup label="Static Languages">
        <option value="cpp">C++</option>
    </optgroup>
    <option value="csharp">C#</option>
</select>
```

Another common requirement when working with `<select>` elements is to include an option in the list that indicates “no value selected,” as shown in figure 8.7. Without this extra option, the default `<select>` dropdown will always have a value, and will default to the first item in the list.



Pick your favourite language!

Without "No Selection" Option

JavaScript

- JavaScript
- C++
- Python
- C#

With "No Selection" Option

Not selected

Figure 8.7 Without a "no selection" option, the `<select>` element will always have a value. This may not be the behavior you desire if you don't want an `<option>` to be selected by default.

You can achieve this in one of two ways: you could either add the "not selected" option to the available `SelectListItem`s, or you could manually add the option to the Razor, for example by using

```
<select asp-for="SelectedValue" asp-items="Model.Items">
    <option Value = "">**Not selected**</option>
</select>
```

This will add an extra `<option>` at the top of your `<select>` element, with a blank `Value` attribute, allowing you to provide a "no selection" option for the user.

> **TIP** Adding a "no selection" option to a `<select>` element is so common; you might want to create a partial view to encapsulate this logic.

With the Input Tag Helper and Select Tag Helper under your belt, you should be able to create most of the forms that you'll need. You have all the pieces you need to create the currency converter application now, with one exception.

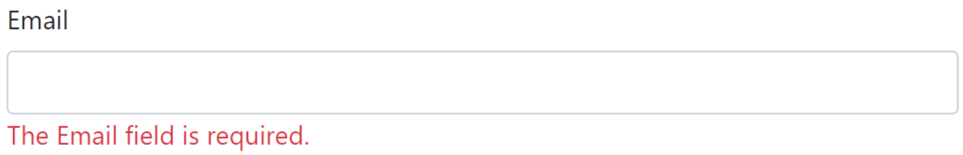
Remember, whenever you accept input from a user, you should always validate the data. The Validation Tag Helpers provide a way to display model validation errors to the user on your form, without having to write a lot of boilerplate markup.

8.2.5 The Validation Message and Validation Summary Tag Helpers

In section 8.2.3 you saw that the Input Tag Helper generates the necessary `data-val-*` validation attributes on form input elements themselves. But you also need somewhere to display the validation messages. This can be achieved for each property in your view model using the Validation Message Tag Helper applied to a `<span>` by using the `asp-validation-for` attribute:

```
<span asp-validation-for="Email"></span>
```

When an error occurs during client-side validation, the appropriate error message for the referenced property will be displayed in the `<span>`, as shown in figure 8.8. This `<span>` element will also be used to show appropriate validation messages if server-side validation fails, and the form is being redisplayed.



The Email field is required.

Figure 8.8 Validation messages can be shown in an associated `<span>` by using the Validation Message Tag Helper.

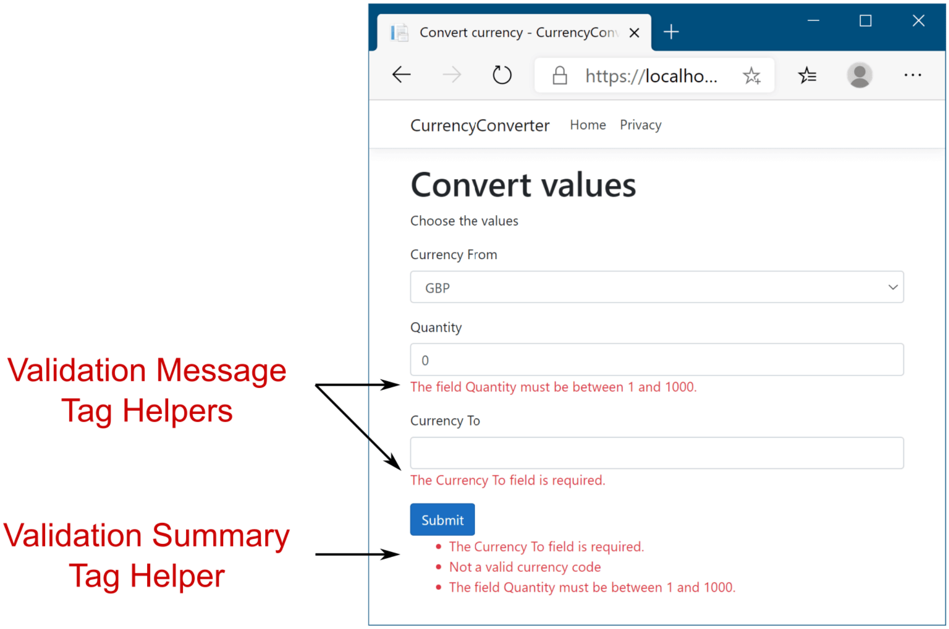
Any errors associated with the `Email` property stored in `ModelState` will be rendered in the element body, and the element will have appropriate attributes to hook into jQuery validation:

```
<span class="field-validation-valid" data-valmsg-for="Email"
  data-valmsg-replace="true">The Email Address field is required.</span>
```

The validation error shown in the element will be replaced when the user updates the `Email` `<input>` field and client-side validation is performed.

> **NOTE** For further details on `ModelState` and server-side model validation, see chapter 6.

As well as displaying validation messages for individual properties, you can also display a summary of all the validation messages in a `<div>` by using the Validation Summary Tag Helper, shown in figure 8.9. This renders a `<ul>` containing a list of the `ModelState` errors.



Convert currency - CurrencyCon
https://localho...
CurrencyConverter
Home
Privacy
Validation Message Tag Helpers
Validation Summary Tag Helper
Submit

Convert values

Choose the values

Currency From: GBP

Quantity: 0

The field Quantity must be between 1 and 1000.

Currency To:

The Currency To field is required.

- The Currency To field is required.
- Not a valid currency code
- The field Quantity must be between 1 and 1000.

Figure 8.9 Form showing validation errors. The Validation Message Tag Helper is applied to `<span>`, close to the associated input. The Validation Summary Tag Helper is applied to a `<div>`, normally at the top or bottom of the form.

The Validation Summary Tag Helper is applied to a `<div>` using the `asp-validation-summary` attribute and providing a `ValidationSummary` enum value, such as

```
<div asp-validation-summary="All"></div>
```

The `ValidationSummary enum` controls which values are displayed, and has three possible values:

- `None`—Don’t display a summary. (I don’t know why you’d use this.)
- `ModelOnly`—Only display errors that are *not* associated with a property.
- `All`—Display errors either associated with a property or with the model.

The Validation Summary Tag Helper is particularly useful if you have errors associated with your page that aren’t specific to a single property. These can be added to the model state by using a blank key, as shown in listing 8.8. In this example, the property validation passed, but we provide additional model-level validation to check that we aren’t trying to convert a currency to itself.

Listing 8.8 Adding model-level validation errors to the `ModelState`

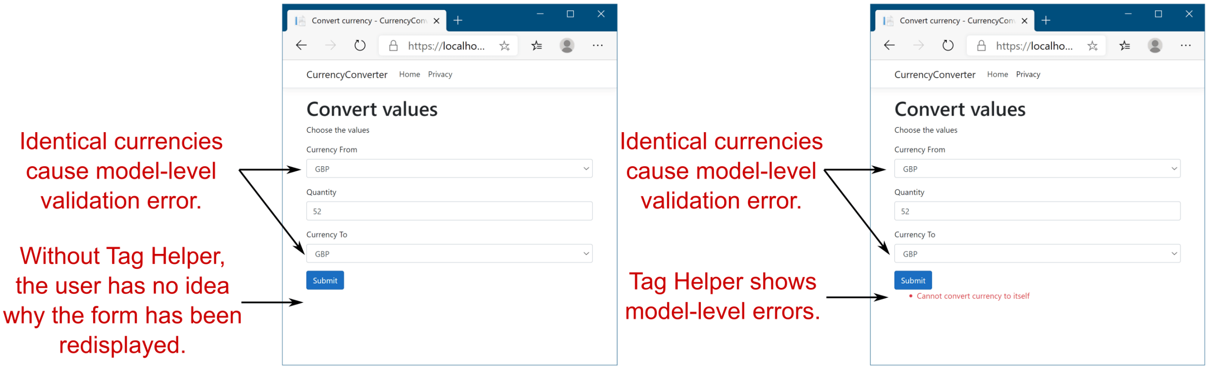
```
public class ConvertModel : PageModel
```

```
{
    [BindProperty]
    public InputModel Input { get; set; }

    [HttpPost]
    public IActionResult OnPost()
    {
        if(Input.CurrencyFrom == Input.CurrencyTo)         #A
        {
            ModelState.AddModelError(                      #B
                string.Empty,                              #B
                "Cannot convert currency to itself");      #B
        }
        if (!ModelState.IsValid)                           #C
        {                                                  #C
            return Page();                                 #C
        }                                                  #C

        //store the valid values somewhere etc
        return RedirectToPage("Checkout");
    }
}
```

#A Can't convert currency to itself
#B Adds model-level error, not tied to a specific property, by using empty key
#C If there are any property-level or model-level errors, display them.

Without the Validation Summary Tag Helper, the model-level error would still be added if the user used the same currency twice, and the form would be redisplayed. Unfortunately, there would have been no visual cue to the user indicating why the form did not submit—obviously that's a problem! By adding the Validation Summary Tag Helper, the model-level errors are shown to the user so they can correct the problem, as shown in figure 8.10.

> **NOTE** For simplicity, I added the validation check to the page handler. A better approach might be to create a custom validation attribute to achieve this instead. That way, your handler stays lean and sticks to the single responsibility principle. You'll see how to achieve this in chapter 19.



Identical currencies cause model-level validation error.

Without Tag Helper, the user has no idea why the form has been redisplayed.

Convert values

Choose the values

Currency From: GBP

Quantity: 52

Currency To: GBP

Identical currencies cause model-level validation error.

Tag Helper shows model-level errors.

Convert values

Choose the values

Currency From: GBP

Quantity: 52

Currency To: GBP

- Cannot convert currency to itself

Figure 8.10 Model-level errors are only displayed by the Validation Summary Tag Helper. Without one, users won't have any indication that there were errors on the form, and so won't be able to correct them.

This section has covered most of the common Tag Helpers available for working with forms, including all the pieces you need to build the currency converter forms. They should give you everything you need to get started building forms in your own applications. But forms aren't the only area in which Tag Helpers are useful; they're generally applicable any time you need to mix server-side logic with HTML generation.

One such example is generating links to other pages in your application using routing-based URL generation. Given that routing is designed to be fluid as you refactor your application, keeping track of the exact URLs the links should point to would be a bit of a maintenance nightmare if you had to do it "by hand." As you might expect, there's a Tag Helper for that: the Anchor Tag Helper.

8.3 Generating links with the Anchor Tag Helper

At the end of chapter 5, I showed how you could generate URLs for links to other pages in your application from inside your page handlers by using `ActionResult`s. Views are the other common place where you need to link to other pages in your application, normally by way of an `<a>` element with an `href` attribute pointing to the appropriate URL.

In this section I show how to use the Anchor Tag Helper to generate the URL for a given Razor Page using routing. Conceptually, this is almost identical to the way the Form Tag Helper generates the `action` URL, as you saw in section 8.2.1. For the most part, using the Anchor Tag Helper is identical too; you provide `asp-page` and `asp-page-handler` attributes, along with `asp-route-*` attributes as necessary. The default Razor Page templates use the Anchor Tag Helper to generate the links shown in the navigation bar using the code in the listing below.

Listing 8.9 Using the Anchor Tag Helper to generate URLs in _Layout.cshtml

```
<ul class="navbar-nav flex-grow-1">
```

```
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-area="" asp-page="/Index">Home</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-area="" asp-page="/Privacy">Privacy</a>
    </li>
</ul>
```

As you can see, each `<a>` element has an `asp-page` attribute. This Tag Helper uses the routing system to generate an appropriate URL for the `<a>`, resulting in the following markup:

```
<ul class="nav navbar-nav">
    <li class="nav-item">
        <a class="nav-link text-dark" href="/">Home</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" href="/Privacy">Privacy</a>
    </li>t
</ul>
```

The URLs use default values where possible, so the `Index` Razor Page generates the simple `"/"` URL instead of `"/Index"`.

If you need more control over the URL generated, the Anchor Tag Helper exposes several additional properties you can set, which will be used during URL generation. The most commonly used with Razor Pages are:

- `asp-page`—Sets the Razor Page to execute.
- `asp-page-handler`—Sets the Razor Page handler to execute.
- `asp-area`—Sets the area route parameter to use. Areas can be used to provide an additional layer of organization to your application.[36]
- `asp-host`—If set, the link will point to the provided host and will generate an absolute URL instead of a relative URL.
- `asp-protocol`—Sets whether to generate an http or https link. If set, it will generate an absolute URL instead of a relative URL.
- `asp-route-*`—Sets the route parameters to use during generation. Can be added multiple times for different route parameters.

By using the Anchor Tag Helper and its attributes, you generate your URLs using the routing system, as described in chapter 5. This reduces the duplication in your code by removing the hardcoded URLs you'd otherwise need to embed in all your views.

[36] I won't cover areas in detail this book. They're an optional aspect of MVC that are often only used on large projects. You can read about them here: http://mng.bz/3X64.

If you find yourself writing repetitive code in your markup, chances are someone has written a Tag Helper to help with it. The Append Version Tag Helper in the following section is a great example of using Tag Helpers to reduce the amount of fiddly code required.

8.4 Cache-busting with the Append Version Tag Helper

A common problem with web development, both when developing and when an application goes into production, is ensuring that browsers are all using the latest files. For performance reasons, browsers often cache files locally and reuse them for subsequent requests, rather than calling your application every time a file is requested.

Normally, this is great—most of the static assets in your site rarely change, so caching them significantly reduces the burden on your server. Think of an image of your company logo—how often does that change? If every page shows your logo, then caching the image in the browser makes a lot of sense.

But what happens if it *does* change? You want to make sure users get the updated assets as soon as they're available. A more critical requirement might be if the JavaScript files associated with your site change. If users end up using cached versions of your JavaScript then they might see strange errors, or your application might appear broken to them.

This conundrum is a common one in web development, and one of the most common ways for handling it is to use a *cache-busting query string*.

> **DEFINITION** A cache-busting query string adds a query parameter to a URL, such as `?v=1`. Browsers will cache the response and use it for subsequent requests to the URL. When the resource changes, the query string is also changed, for example to `?v=2`. Browsers will see this as a request for a new resource and will make a fresh request.

The biggest problem with this approach is that it requires you to update a URL every time an image, CSS, or JavaScript file changes. This is a manual step that requires updating every place the resource is referenced, so it's inevitable that mistakes are made. Tag Helpers to the rescue! When you add a `<script>`, `<img>`, or `<link>` element to your application, you can use Tag Helpers to automatically generate a cache-busting query string:

```
<script src="~/js/site.js" asp-append-version="true"></script>
```

The `asp-append-version` attribute will load the file being referenced and generate a unique hash based on its contents. This is then appended as a unique query string to the resource URL:

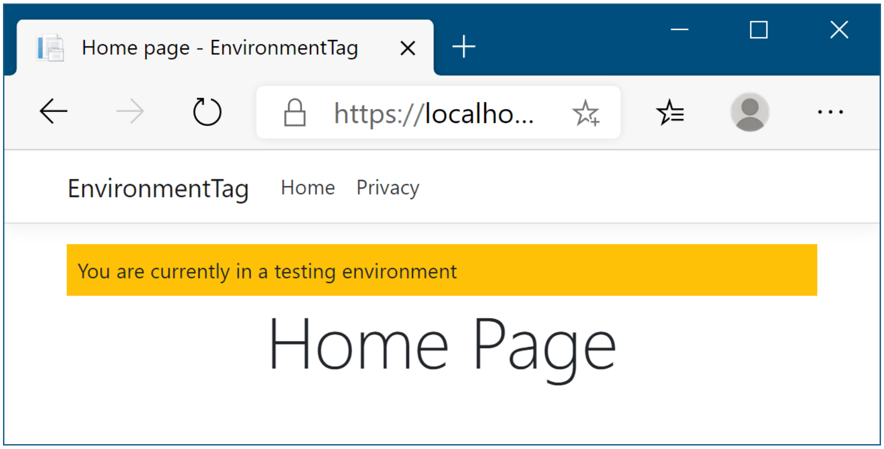
```
<script src="/js/site.js?v=EWaMeWsJBYWmL2g_KkgXZQ5nPe"></script>
```

As this value is a hash of the file contents, it will remain unchanged as long as the file isn't modified, so the file will be cached in users' browsers. But if the file is modified, then the hash of the contents will change and so will the query string. This ensures browsers are always served the most up-to-date files for your application without your having to worry about manually updating every URL whenever you change a file.

So far in this chapter you've seen how to use Tag Helpers for forms, link generation, and cache busting. You can also use Tag Helpers to conditionally render different markup depending on the current environment. This uses a technique you haven't seen yet, where the Tag Helper is declared as a completely separate element.

8.5 Using conditional markup with the Environment Tag Helper

In many cases, you want to render different HTML in your Razor templates depending if your website is running in a development or production environment. For example, in development, you typically want your JavaScript and CSS assets to be verbose and easy to read, but in production you'd process these files to make them as small as possible. Another example might be the desire to apply a banner to the application when it's running in a testing environment, which is removed when you move to production, shown in figure 8.11.



You are currently in a testing environment

Figure 8.11 The warning banner will be shown whenever you're running in a testing environment, to make it easy to distinguish from production.

NOTE You'll learn about configuring your application for multiple environments in chapter 11.

You've already seen how to use C# to add `if` statements to your markup, so it would be perfectly possible to use this technique to add an extra `div` to your markup when the current environment has a given value. If we assume that the `env` variable contains the current environment, then you could use something like

```
@if(env == "Testing" || env == "Staging")
{
    <div class="warning">You are currently on a testing environment</div>
}
```

There's nothing wrong with this, but a better approach would be to use the Tag Helper paradigm to keep your markup clean and easy to read. Luckily, ASP.NET Core comes with the `EnvironmentTagHelper`, which can be used to achieve the same result in a slightly clearer way:

```
<environment include="Testing,Staging">
    <div class="warning">You are currently on a testing environment</div>
</environment>
```

This Tag Helper is a little different from the others you've seen before. Instead of augmenting an existing HTML element using an `asp-` attribute, *the whole element* is the Tag Helper. This Tag Helper is completely responsible for generating the markup, and it uses an attribute to configure it.

Functionally, this Tag Helper is identical to the C# markup (although for now I've glossed over how the `env` variable could be found), but it's more declarative in its function than the C# alternative. You're obviously free to use either approach, but personally I like the HTML-like nature of Tag Helpers.

We've reached the end of this chapter on Tag Helpers, and with it, our first look at building traditional web applications that display HTML to users. In the last part of the book, we'll revisit Razor templates, and you'll learn how to build custom components like custom Tag Helpers and View Components. For now, you have everything you need to build complex Razor layouts—the custom components can help tidy up your code down the line.

Part 1 of this book has been a whistle-stop tour of how to build Razor Page applications with ASP.NET Core. You now have the basic building blocks to start making simple ASP.NET Core applications. In the second part of this book, I'll show you some of the additional features you'll need to understand to build complete applications. But before we get to that, I'll take a chapter to discuss building Web APIs.

I've mentioned the Web API approach previously, in which your application serves data using the MVC framework, but instead of returning user-friendly HTML, it returns machine-friendly JSON. In the next chapter, you'll see why and how to build a Web API, take a look at an alternative routing system designed for APIs, and learn how to generate JSON responses to requests.

8.6 Summary

- With Tag Helpers, you can bind your data model to HTML elements, making it easier to generate dynamic HTML while remaining editor friendly.
- As with Razor in general, Tag Helpers are for server-side rendering of HTML only. You can't use them directly in frontend frameworks, such as Angular or React.
- Tag Helpers can be standalone elements or can attach to existing HTML using attributes. This lets you both customise HTML elements and add entirely new elements.
- Tag Helpers can customize the elements they're attached to, add additional attributes, and customize how they're rendered to HTML. This can greatly reduce the amount of markup you need to write.

- Tag Helpers can expose multiple attributes on a single element. This makes it easier to configure the Tag Helper, as you can set multiple, separate, values.
- You can add the `asp-page` and `asp-page-handler` attributes to the `<form>` element to set the `action` URL using the URL generation feature of Razor Pages.
- You specify route values to use during routing with the Form Tag Helper using `asp-route-*` attributes. These values are used to build the final URL or are passed as query data.
- The Form Tag Helper also generates a hidden field that you can use to prevent CSRF attacks. This is added automatically and is an important security measure.
- You can attach the Label Tag Helper to a `<label>` using `asp-for`. It generates an appropriate `for` attribute and caption based on the `[Display] Data-Annotations` attribute and the `PageModel` property name.
- The Input Tag Helper sets the `type` attribute of an `<input>` element to the appropriate value based on a bound property's `Type` and any `DataAnnotations` applied to it. It also generates the `data-val-*` attributes required for client-side validation. This significantly reduces the amount of HTML code you need to write.
- To enable client-side validation, you must add the necessary JavaScript files to your view for jQuery validation and unobtrusive validation.
- The Select Tag Helper can generate drop-down `<select>` elements as well as list boxes, using the `asp-for` and `asp-items` attributes. To generate a multiselect `<select>` element, bind the element to an `IEnumerable` property on the view model. You can use these approaches to generate several different styles of select box.
- The items supplied in `asp-for` must be an `IEnumerable<SelectListItem>`. If you try to bind another type, you'll get a compile time error in your Razor view.
- You can generate an `IEnumerable<SelectListItem>` for an `enum TEnum` using the `Html.GetEnumSelectList<TEnum>()` helper method. This saves you having to write the mapping code yourself.
- The Select Tag Helper will generate `<optgroup>` elements if the items supplied in `asp-for` have an associated `SelectListGroup` on the `Group` property. Groups are can be used to separate items in select lists.
- Any extra additional `<option>` elements added to the Razor markup will be passed through to the final HTML. You can use these additional elements to easily add a "no selection" option to the `<select>` element.
- The Validation Message Tag Helper is used to render the client- and server-side validation error messages for a given property. This gives important feedback to your users when elements have errors. Use the `asp-validation-for` attribute to attach the Validation Message Tag Helper to a `<span>`.
- The Validation Summary Tag Helper is used to display validation errors for the model, as well as for individual properties. You can use model-level properties to display additional validation that doesn't apply to just one property. Use the asp`-validation-summary` attribute to attach the Validation Summary Tag Helper to a `<div>`.

- You can generate `<a>` URLs using the Anchor Tag Helper. This Helper uses routing to generate the `href` URL using `asp-page`, `asp-page-handler`, and `asp-route-*` attributes, giving you the full power of routing.
- You can add the `asp-append-version` attribute to `<link>`, `<script>`, and `<img>` elements to provide cache-busting capabilities based on the file's contents. This ensures users cache files for performance reasons, yet still always get the latest version of files.
- You can use the Environment Tag Helper to conditionally render different HTML based on the app's current execution environment. You can use this to render completely different HTML in different environments if you wish.

9

Creating a Web API for mobile and client applications using MVC

This chapter covers

- **Creating a Web API controller to return JSON to clients**
- **Using attribute routing to customize your URLs**
- **Generating a response using content negotiation**
- **Applying common conventions with the `[ApiController]` attribute**

In the previous five chapters, you've worked through each layer of a server-side rendered ASP.NET Core application, using Razor Pages to render HTML to the browser. In this chapter, you'll see a different take on an ASP.NET Core application. Instead of using Razor Pages, we'll explore Web APIs, which serve as the backend for client-side SPAs and mobile apps.

You can apply much of what you've already learned to Web APIs; they use the same MVC design pattern, and the concepts of routing, model binding, and validation all carry through. The differentiation from traditional web applications is primarily in the *view* part of MVC. Instead of returning HTML, they return data as JSON or XML, which client applications use to control their behavior or update the UI.

In this chapter, you'll learn how to define controllers and actions and see how similar they are to the Razor Pages and controllers you already know. You'll learn how to create an API model to return data and HTTP status codes in response to a request, in a way that client apps can understand.

After exploring how the MVC design pattern applies to Web APIs, you'll see how a related topic works with Web APIs: routing. We'll look at how *attribute routing* reuses many of the same concepts from chapter 5 but applies them to your action methods rather than to Razor Pages.

One of the big features added in ASP.NET Core 2.1 was the `[ApiController]` attribute. This attribute applies several common conventions used in Web APIs, which reduces the amount of code you must write yourself. In section 9.5 you'll learn how automatic bad responses for invalid requests, model binding parameter inference, and `ProblemDetails` objects can make building APIs easier and more consistent.

You'll also learn how to format the API models returned by your action methods using content negotiation, to ensure you generate a response that the calling client can understand. As part of this, you'll learn how to add support for additional format types, such as XML, so that you can generate XML responses if the client requests it.

One of the great aspects of ASP.NET Core is the variety of applications you can create with it. The ability to easily build a generalized HTTP Web API presents the possibility of using ASP.NET Core in a greater range of situations than can be achieved with traditional web apps alone. But should *you* build a Web API and why? In the first section of this chapter, I'll go over some of the reasons why you might or might not want to create a Web API.

9.1 What is a Web API and when should you use one?

Traditional web applications handle requests by returning HTML to the user, which is displayed in a web browser. You can easily build applications of this nature using Razor Pages to generate HTML with Razor templates, as you've seen in recent chapters. This approach is common and well understood, but the modern application developer also has a number of other possibilities to consider, as shown in figure 9.1.

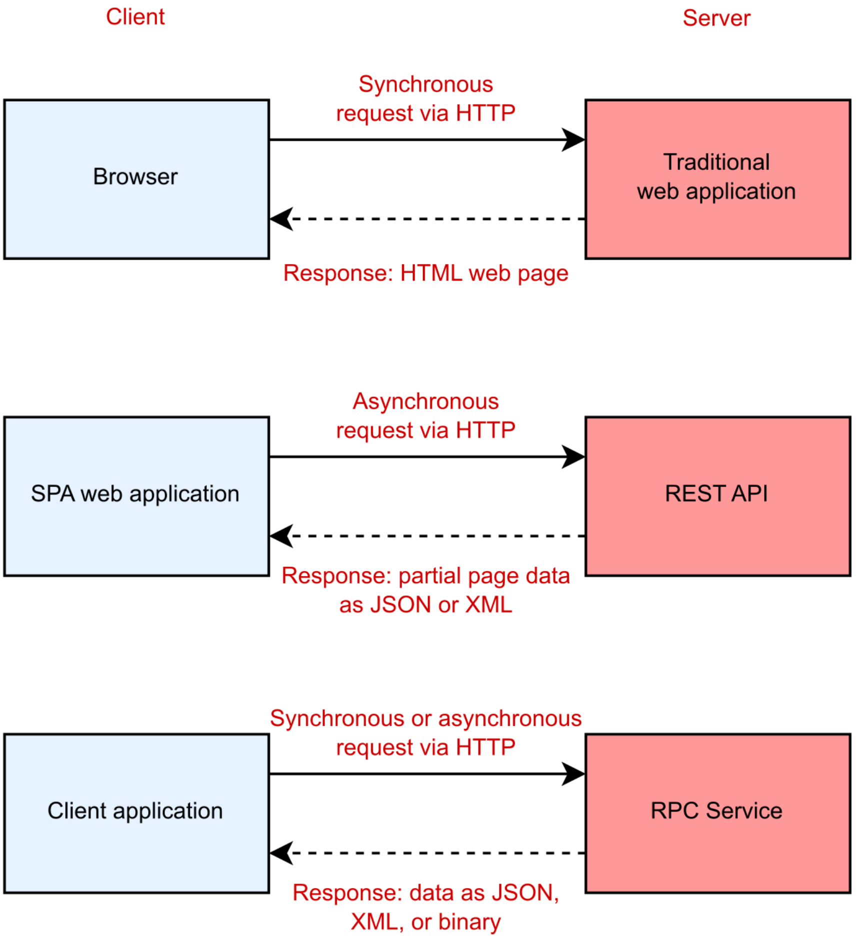

Client
Server
Synchronous request via HTTP
Browser
Traditional web application
Response: HTML web page
Asynchronous request via HTTP
SPA web application
REST API
Response: partial page data as JSON or XML
Synchronous or asynchronous request via HTTP
Client application
RPC Service
Response: data as JSON, XML, or binary

Figure 9.1 Modern developers have to consider a number of different consumers of their applications. As well as traditional users with web browsers, these could be SPAs, mobile applications, or other apps.

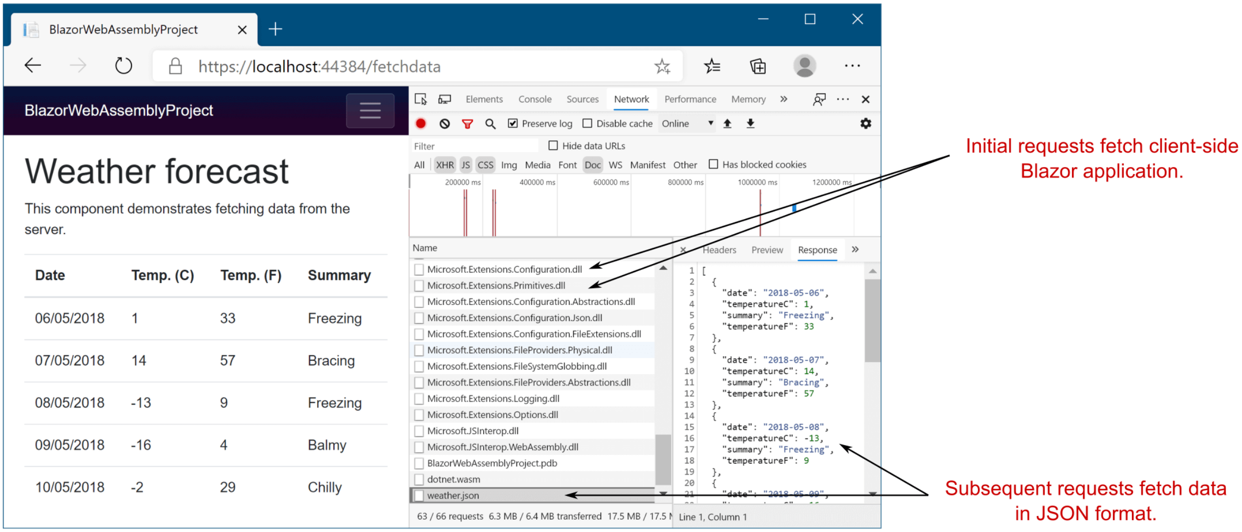
Client-side single-page applications (SPAs) have become popular in recent years with the development of frameworks such as Angular, React, and Vue. These frameworks typically use JavaScript that runs in a user's web browser to generate the HTML they see and interact with. The server sends this initial JavaScript to the browser when the user first reaches the app. The user's browser loads the JavaScript and initializes the SPA, before loading any application data from the server.

> **NOTE** Blazor WebAssembly is an exciting new SPA framework. Blazor lets you write an SPA that runs in the browser just like other SPAs, but it uses C# and Razor templates instead of JavaScript by using the new web

standard, WebAssembly. I don't cover Blazor in this book, so to find out more, I recommend *Blazor in Action*, by Chris Sainty (Manning, 2021).

Once the SPA is loaded in the browser, communication with a server still occurs over HTTP, but instead of sending HTML directly to the browser in response to requests, the server-side application sends data (normally in a format such as JSON) to the client-side application. The SPA then parses the data and generates the appropriate HTML to show to a user, as shown in figure 9.2. The server-side application endpoint that the client communicates with is sometimes called a *Web API*.

DEFINITION **A *Web API* exposes a number of URLs that can be used to access or change data on a server. It's typically accessed using HTTP.**


BlazorWebAssemblyProject
https://localhost:44384/fetchdata
BlazorWebAssemblyProject
Elements Console Sources Network Performance Memory
Preserve log
Disable cache
Online
Filter
Hide data URLs
All XHR JS CSS Img Media Font Doc WS Manifest Other
Has blocked cookies
200000 ms 400000 ms 600000 ms 800000 ms 1000000 ms 1200000 ms
Name
Headers Preview Response
63 / 66 requests 6.3 MB / 6.4 MB transferred 17.5 MB / 17.5 M
Line 1, Column 1

Weather forecast

This component demonstrates fetching data from the server.

Date	Temp. (C)	Temp. (F)	Summary
06/05/2018	1	33	Freezing
07/05/2018	14	57	Bracing
08/05/2018	-13	9	Freezing
09/05/2018	-16	4	Balmy
10/05/2018	-2	29	Chilly

```
Microsoft.Extensions.Configuration.dll
Microsoft.Extensions.Primitives.dll
Microsoft.Extensions.Configuration.Abstractions.dll
Microsoft.Extensions.Configuration.Json.dll
Microsoft.Extensions.Configuration.FileExtensions.dll
Microsoft.Extensions.FileProviders.Physical.dll
Microsoft.Extensions.FileSystemGlobbing.dll
Microsoft.Extensions.FileProviders.Abstractions.dll
Microsoft.Extensions.Logging.dll
Microsoft.Extensions.Options.dll
Microsoft.JSInterop.dll
Microsoft.JSInterop.WebAssembly.dll
BlazorWebAssemblyProject.pdb
dotnet.wasm
weather.json
```

```
[
  {
    "date": "2018-05-06",
    "temperatureC": 1,
    "summary": "Freezing",
    "temperatureF": 33
  },
  {
    "date": "2018-05-07",
    "temperatureC": 14,
    "summary": "Bracing",
    "temperatureF": 57
  },
  {
    "date": "2018-05-08",
    "temperatureC": -13,
    "summary": "Freezing",
    "temperatureF": 9
  },
  {
    "date": "2018-05-09",
```

Initial requests fetch client-side Blazor application.

Subsequent requests fetch data in JSON format.

Figure 9.2 A sample client-side SPA using Blazor WebAssembly. The initial requests load the SPA files into the browser, and subsequent requests fetch data from a Web API, formatted as JSON.

These days, mobile applications are common and are, from the server application's point of view, similar to client-side SPAs. A mobile application will typically communicate with a server application using an HTTP Web API, receiving data in a common format, such as JSON, just like an SPA. It then modifies the application's UI depending on the data it receives.

One final use case for a Web API is where your application is designed to be partially or solely consumed by other back-end services. Imagine you've built a web application to send emails. By creating a Web API, you can allow other application developers to use your email service by sending you an email address and a message. Virtually all languages and platforms have access to an HTTP library they could use to access your service from code.

This is all there is to a Web API. It exposes a number of endpoints (URLs) that client applications can send requests to and retrieve data from. These are used to power the behavior of the client apps, as well as to provide all the data the client apps need to display the correct interface to a user.

Whether you need or want to create a Web API for your ASP.NET Core application depends on the type of application you want to build. If you're familiar with client-side frameworks, will need to develop a mobile application, or already have an SPA build-pipeline configured, then you'll most likely want to add Web APIs for them to be able to access your application.

One of the selling points of using a Web API is that it can serve as a generalized backend for all of your client applications. For example, you could start by building a client-side application that uses a Web API. Later, you could add a mobile app that uses the same Web API, with little or no modification required to your ASP.NET Core code.

If you're new to web development, have no need to call your application from outside a web browser, or don't want/need the effort involved in configuring a client-side application, then you probably won't need Web APIs initially. You can stick to generating your UI using Razor Pages and will no doubt be highly productive!

> **NOTE** Although there has definitely been a shift toward client-side frameworks, server-side rendering using Razor is still relevant. Which approach you choose will depend largely on your preference for building HTML applications in the traditional manner versus using JavaScript on the client.

Having said that, adding Web APIs to your application isn't something you have to worry about ahead of time. Adding them later is simple, so you can always ignore them initially and add them in as the need arises. In many cases, this will be the best approach.

SPAs with ASP.NET Core

The cross-platform and lightweight design of ASP.NET Core means it lends itself well to acting as a backend for your SPA framework of choice. Given the focus of this book and the broad scope of SPAs in general, I won't be looking at Angular, React, or other SPAs here. Instead, I suggest checking out the resources appropriate to your chosen SPA. Books are available from Manning for all the common client-side JavaScript frameworks:

- React in Action by Mark Tielens Thomas (Manning, 2018) https://livebook.manning.com/book/react-in-action/.
- Angular in Action by Jeremy Wilken (Manning, 2018) https://livebook.manning.com/book/angular-in-action/.
- Vue.js in Action by Erik Hanchett with Benjamin Listwon (Manning, 2018) https://livebook.manning.com/book/vue-js-in-action/.

To learn about Blazor WebAssembly, see the documentation at https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor.

Once you've established that you need a Web API for your application, creating one is easy, as it's built in to ASP.NET Core. In the next section, you'll see how to create a Web API project and your first API controller.

9.2 Creating your first Web API project

In this section you'll learn how to create an ASP.NET Core Web API project and create your first Web API controllers. You'll see how to use controller action methods to handle HTTP requests, and how to use `ActionResult`s to generate a response.

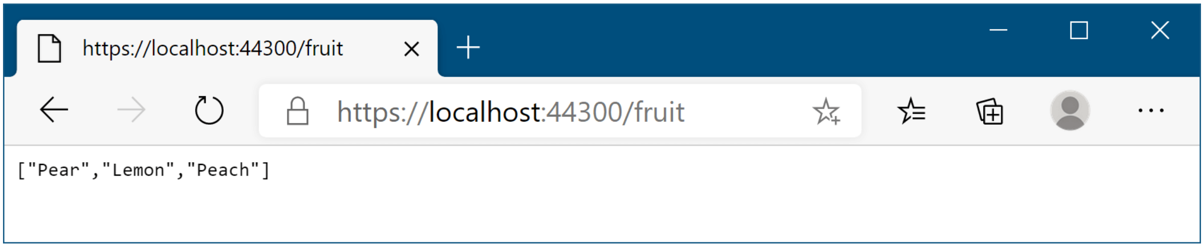
Some people think of the MVC design pattern as only applying to applications that directly render their UI, like the Razor views you've seen in previous chapters. In ASP.NET Core, the MVC pattern applies equally well when building a Web API; but the *view* part of MVC involves generating a *machine*-friendly response rather than a *user*-friendly response.

As a parallel to this, you create Web API controllers in ASP.NET Core in the very same way you create traditional MVC controllers. The only thing that differentiates them from a code perspective is the type of data they return—MVC controllers typically return a `ViewResult`; Web API controllers generally return raw .NET objects from the action methods, or an `IActionResult` such as `StatusCodeResult`, as you saw in chapter 4.

> **NOTE** This is different from the previous version of ASP.NET, where the MVC and Web API stacks were completely independent. ASP.NET Core unifies the two stacks into a single approach (and adds Razor Pages to the mix) which makes using any option in a project painless!

To give you an initial taste of what you're working with, figure 9.3 shows the result of calling a Web API endpoint from your browser. Instead of a friendly HTML UI, you receive data that can be easily consumed in code. In this example, the Web API returns a list of `string` fruit names as JSON when you request the URL `/fruit`.

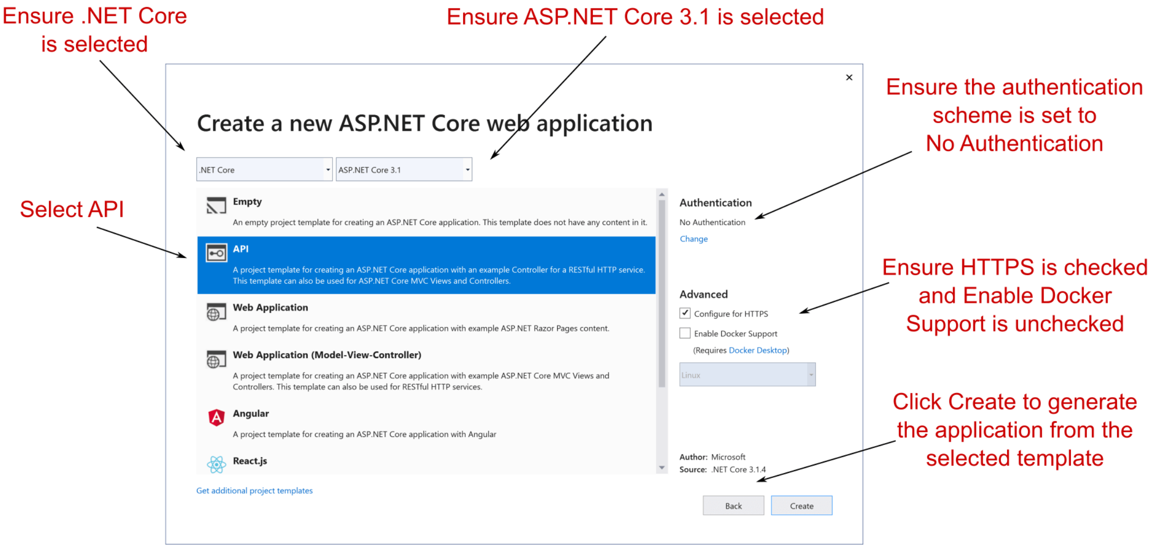
> **TIP** Web APIs are normally accessed from code by SPAs or mobile apps, but by accessing the URL in your web browser directly, you can view the data the API is returning.



```
["Pear","Lemon","Peach"]
```

Figure 9.3 Testing a Web API by accessing the URL in the browser. A `GET` request is made to the /fruit URL, which returns a `List<string>` that has been JSON-encoded into an array of strings.

You can create a new Web API project in Visual Studio using the same New Project process you saw in chapter 2. Create a new ASP.NET Core application, providing a project name, and, when you reach the New Project dialog, select the API template, as shown in figure 9.4. If you're using the CLI, you can create a similar template using `dotnet new webapi -o WebApplication1`.



Ensure .NET Core is selected

Ensure ASP.NET Core 3.1 is selected

Ensure the authentication scheme is set to No Authentication

Select API

Ensure HTTPS is checked and Enable Docker Support is unchecked

Click Create to generate the application from the selected template

Create a new ASP.NET Core web application

.NET Core

ASP.NET Core 3.1

Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Web Application
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

Web Application (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Angular
A project template for creating an ASP.NET Core application with Angular

React.js

Get additional project templates

Authentication
No Authentication
Change

Advanced
☑ Configure for HTTPS
□ Enable Docker Support
(Requires Docker Desktop)

Linux

Author: Microsoft
Source: .NET Core 3.1.4

Figure 9.4 The web application template screen. This screen follows on from the configure your project dialog and lets you customize the template that will generate your application. Choose the API template to create a Web API project.

The API template configures the ASP.NET Core project for Web API controllers only. This configuration occurs in the Startup.cs file, as shown in listing 9.1. If you compare this template to your Razor Pages projects, you'll see that the API project uses `AddControllers()` instead of `AddRazorPages()` in the `ConfigureServices` method. Also, the API controllers are added instead of Razor Pages by calling `MapControllers()` in the `UseEndpoints` method. If you're using both Razor Pages and Web APIs in a project, you'll need to add all these method calls to your application.

Listing 9.1 The Startup class for the default Web API project

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();                              #A
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseHttpsRedirection();
        app.UseRouting();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();                            #B
        });
    }
}
```

#A AddControllers adds the necessary services for API controllers to your application
#B MapControllers configures the API controller actions in your app as endpoints

The Startup.cs file in listing 9.1 instructs your application to find all API controllers in your application and to configure them in the `EndpointMiddleware`. Each action method becomes an endpoint and can receive requests when the `RoutingMiddleware` maps an incoming URL to the action method.

> **NOTE** **You don't have to use separate projects for Razor Pages and Web API controllers, but I prefer to do so where possible. There are certain aspects (such as error handling and authentication) that are made easier by taking this approach. Of course, running two separate applications has its own difficulties!**

You can add a Web API controller to your project be creating a new .cs file anywhere in your project. Traditionally these are placed in a folder called Controllers, but that's not a technical requirement. Listing 9.2 shows the code that was used to create the Web API demonstrated in figure 9.3. This trivial example highlights the similarity to traditional MVC controllers.

Listing 9.2 A simple Web API controller

```
[ApiController]                                    #A
public class FruitController : ControllerBase      #B
{
    List<string> _fruit = new List<string>         #C
    {                                              #C
        "Pear",                                    #C
        "Lemon",                                   #C
        "Peach"                                    #C
    };                                             #C

    [HttpGet("fruit")]                             #D
    public IEnumerable<string> Index()             #E
    {                                              #F
        return _fruit;                             #F
    }                                              #F
}
```

#A Web API controllers typically use the [ApiController] attribute to opt-in to common conventions
#B The ControllerBase class provides several useful functions for creating IActionResults

#C The data returned would typically be fetched from the application model in a real app.
#D The [HttpGet] attribute defines the route template used to call the action
#E The name of the action method, Index, isn't used for routing. It can be anything you like.
#F The controller exposes a single action method which returns the list of fruit.

Web APIs typically use the `[ApiController]` attribute (introduced in .NET Core 2.1) on API controllers and derive from the `ControllerBase` class. The base class provides several helper methods for generating results, and the `[ApiController]` attribute automatically applies some common conventions, as you'll see in section 9.5.

> **TIP** **There is also a `Controller` base class, which is typically used when you use MVC controllers with Razor views. That's not necessary for Web API controllers, so `ControllerBase` is the better option.**

In listing 9.2 you can see that the action method, `Index`, returns a list of strings directly from the action method. When you return data from an action like this, you're providing the *API model* for the request. The client will receive this data. It's formatted into an appropriate response, a JSON representation of the list in the case of figure 9.3, and sent back to the browser with a `200 OK` status code.

> **TIP** **ASP.NET Core formats data as JSON by default. You'll see how to format the returned data in other ways in section 9.6.**

The URL at which a Web API controller action is exposed is handled in the same way as for traditional MVC controllers and Razor Pages—using routing. The `[HttpGet("fruit")]` attribute applied to the `Index` method indicates the method should use the route template `"fruit"` and should respond to HTTP `GET` requests. You'll learn more about attribute routing in section 9.5.

In listing 9.2, data is returned directly from the action method, but you don't *have* to do that. You're free to return an `ActionResult` instead, and often this is required. Depending on the desired behavior of your API, you may sometimes want to return data, and other times you may want to return a raw HTTP status code, indicating whether the request was successful. For example, if an API call is made requesting details of a product that does not exist, you might want to return a `404 Not Found` status code.

Listing 9.3 shows an example of just such a case. It shows another action on the same `FruitController` as before. This method exposes a way for clients to fetch a specific fruit by an `id`, which we'll assume for this example is its index in the list of `_fruit` you defined in the previous listing. Model binding is used to set the value of the `id` parameter from the request.

> **NOTE** **API controllers use model binding, as you saw in chapter 6, to bind action method parameters to the incoming request. Model binding and validation work in exactly the same way as for Razor Pages, you can bind the request to simple primitives, as well as complex C# objects. The only difference is that there isn't a `PageModel` with `[BindProperty]` properties—you can *only* bind to action method parameters.**

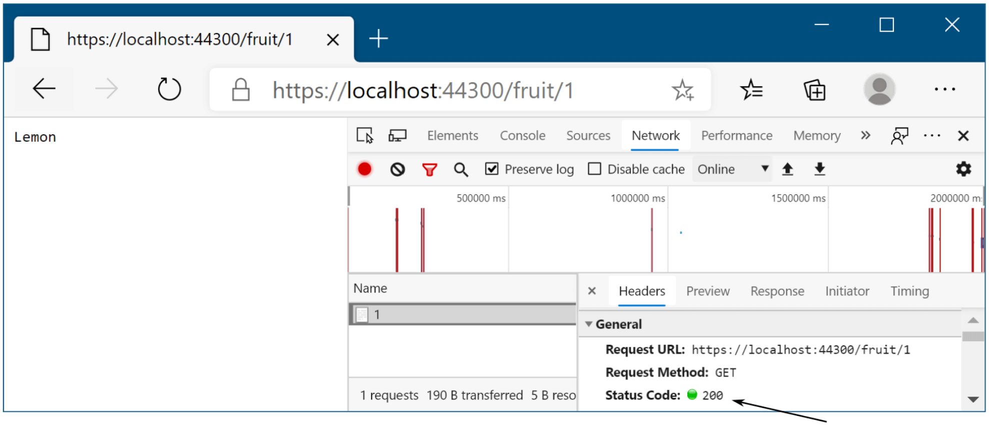
Listing 9.3 A Web API action returning `IActionResult` to handle error conditions

```
[HttpGet("fruit/{id}")]                             #A
public ActionResult<string> View(int id)            #B
{
    if (id >= 0 && id < _fruit.Count)               #C
    {
        return _fruit[id];                           #D
    }
    return NotFound();                              #E
}
```

#A Defines the route template for the action method
#B The action method returns an ActionResult<string>, so it can return a string or an ActionResult.
#C An element can only be returned if the id value is a valid _fruit element index.
#D Returning the data directly will return the data with a 200 status code.
#D NotFound returns a NotFoundResult, which will send a 404 status code.

In the successful path for the action method, the `id` parameter has a value greater than zero and less than the number of elements in `_fruit`. When that's true, the value of the element is returned to the caller. As in listing 9.2, this is achieved by simply returning the data directly, which generates a 200 status code and returns the element in the response body, as shown in figure 9.5. You could also have returned the data using an `OkResult`, using the `Ok` helper method[37] on the `ControllerBase` class—under the hood the result is identical.

[37] Some people get uneasy when they see the phrase "helper method," but there's nothing magic about the `ControllerBase` helpers—they're shorthand for creating a new `IActionResult` of a given type. You don't have to take my word for it though, you can always view the source code for the base class on GitHub at https://github.com/dotnet/aspnetcore/blob/v3.1.3/src/Mvc/Mvc.Core/src/ControllerBase.cs.



Lemon

Request URL: https://localhost:44300/fruit/1
Request Method: GET
Status Code: ● 200

The data is returned in the response body.

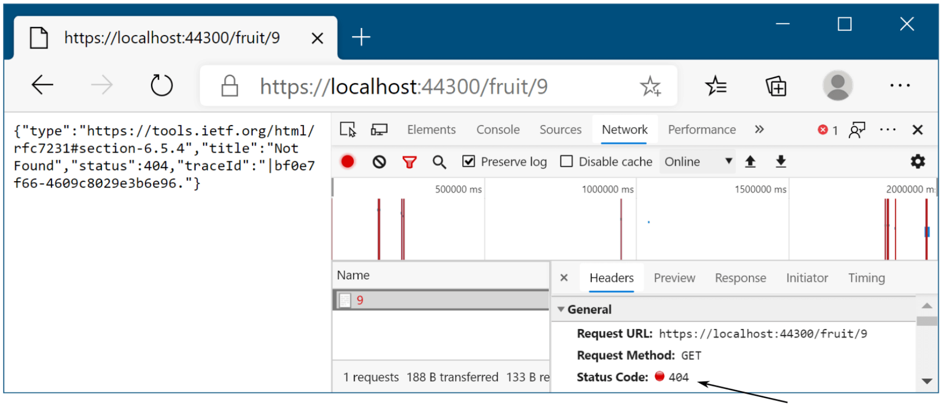
The response is sent with a 200 OK status code.

Figure 9.5 Data returned from an action method is serialized into the response body, and generates a response with status code 200 OK.

If the id is outside the bounds of the _fruit list, then the method calls NotFound to create a NotFoundResult. When executed, this method generates a 404 Not Found status code response. The [ApiController] attribute automatically converts the response into a standard ProblemDetails instance, as shown in figure 9.6.

> **DEFINITION** ProblemDetails **is a web specification for providing machine-readable errors for HTTP APIs. You'll learn more about them in section 9.5.**

The [ApiResponse] attribute generates a Problem Details JSON as the response body.



```
{"type":"https://tools.ietf.org/html/rfc7231#section-6.5.4","title":"Not Found","status":404,"traceId":"|bf0e7f66-4609c8029e3b6e96."}
```

Request URL: https://localhost:44300/fruit/9

Request Method: GET

Status Code: ● 404

The response is sent with a 404 Not Found status code.

Figure 9.6 The `[ApiController]` attribute converts error responses (in this case a 404 response) into the standard `ProblemDetails` format.

One aspect you might find confusing from listing 9.3 is that for the successful case, we return a `string`, but the method signature of `View` says we return an `ActionResult<string>`. How is that possible? Why isn't it a compiler error?

The generic `ActionResult<T>` uses some fancy C# gymnastics with implicit conversions to make this possible[38]. Using `ActionResult<T>` has two benefits:

- You can return either an instance of `T` *or* an `ActionResult` implementation like `NotFoundResult` from the same method. This can be convenient, as in listing 9.3.
- It enables better integration with ASP.NET Core's OpenAPI support (also called Swagger).[39]

You're free to return any type of `ActionResult` from your Web API controllers, but you'll commonly return `StatusCodeResult` instances, which set the response to a specific status code, with or without associated data. `NotFoundResult` and `OkResult` both derive from `StatusCodeResult`, for example. Another commonly used status code is `400 Bad Request`, which is normally returned when the data provided in the request fails validation. This can be generated using a `BadRequestResult`. In many cases the `[ApiController]` attribute can automatically generate `400` responses for you, as you'll see in section 9.5.

[38] You can see how this is achieved in the source code: https://github.com/dotnet/aspnetcore/blob/v3.1.3/src/Mvc/Mvc.Core/src/ActionResultOfT.cs.

[39] OpenAPI is a way of documenting your API controllers, so that you can automatically generate clients for interacting with the API. To learn more, see the documentation at https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger.

TIP You learned about various `ActionResult`s in chapter 4. `BadRequestResult`, `OkResult`, and `NotFoundResult` all inherit from `StatusCodeResult`, and set the appropriate status code for their type (200, 404, and 400, respectively). Using these wrapper classes makes the intention of your code clearer than relying on other developers to understand the significance of the various status code numbers.

Once you've returned an `ActionResult` (or other object) from your controller, it's serialized to an appropriate response. This works in several ways, depending on

- The formatters that your app supports
- The data you return from your method
- The data formats the requesting client can handle

You'll learn more about formatters and serializing data in section 9.6, but before we go any further, it's worth zooming out a little, and exploring the parallels between traditional server-side rendered applications and Web API endpoints. The two are similar, so it's important to establish the patterns that they share and where they differ.

9.3 Applying the MVC design pattern to a Web API

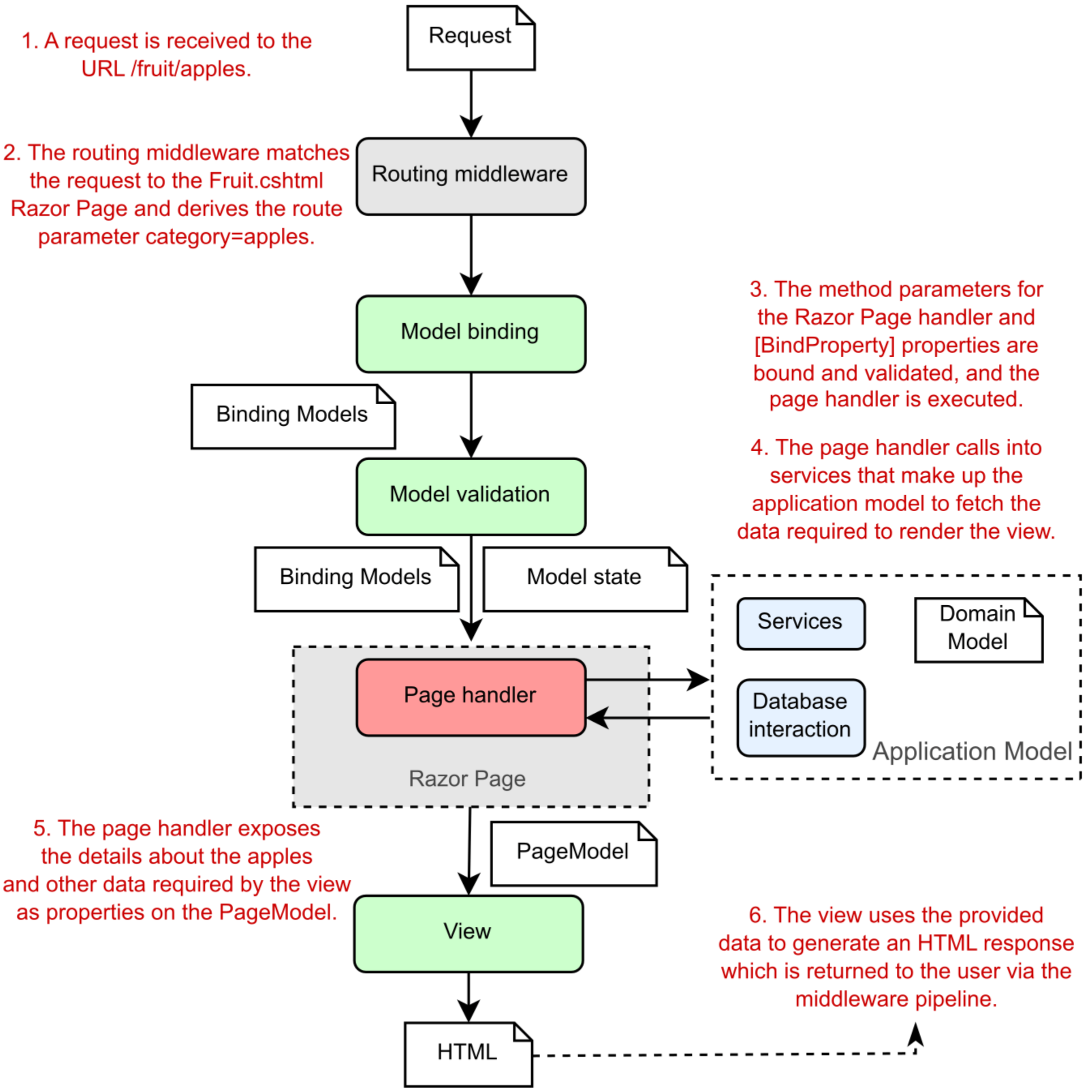
In the previous version of ASP.NET, Microsoft commandeered the generic term "Web API" to create the ASP.NET Web API framework. This framework, as you might expect, was used to create HTTP endpoints that could return formatted JSON or XML in response to requests.

The ASP.NET Web API framework was completely separate from the MVC framework, even though it used similar objects and paradigms. The underlying web stacks for them were completely different beasts and couldn't interoperate.

In ASP.NET Core, that all changed. You now have a *single* framework which you can use to build both traditional web applications and Web APIs. The same underlying framework is used in conjunction with Web API controllers, Razor Pages, and MVC controllers with views. You've already seen this yourself; the Web API `FruitController` you created in section 9.2 look very similar to the MVC controllers you've seen fleetingly in previous chapters.

Consequently, even if you're building an application that consists entirely of Web APIs, using no server-side rendering of HTML, the MVC design pattern still applies. Whether you're building traditional web applications or Web APIs, you can structure your application virtually identically.

After five chapters of it you're, I hope, nicely familiar with how ASP.NET Core handles a request. But just in case you're not, figure 9.7 shows how the framework handles a typical Razor Pages request after passing through the middleware pipeline. This example shows how a request to view the available fruit on a traditional grocery store website might look.



1. A request is received to the URL /fruit/apples.
Request
2. The routing middleware matches the request to the Fruit.cshtml Razor Page and derives the route parameter category=apples.
Routing middleware
3. The method parameters for the Razor Page handler and [BindProperty] properties are bound and validated, and the page handler is executed.
Model binding
Binding Models
Model validation
4. The page handler calls into services that make up the application model to fetch the data required to render the view.
Binding Models
Model state
Services
Domain Model
Database interaction
Application Model
Page handler
Razor Page
5. The page handler exposes the details about the apples and other data required by the view as properties on the PageModel.
PageModel
View
6. The view uses the provided data to generate an HTML response which is returned to the user via the middleware pipeline.
HTML

Figure 9.7 Handling a request to a traditional Razor Pages application, in which the view generates an HTML response that's sent back to the user. This diagram should be very familiar by now!

The `RoutingMiddleware` routes the request to view all the fruit listed in the `apples` category to the Fruit.cshtml Razor Page. The `EndpointMiddleware` then constructs a binding model, validates it, sets it as a property on the Razor Page's `PageModel`, and sets the `ModelState` property on the `PageModel` base class with details of any validation errors. The page handler interacts with the application model by calling into services, talking to a database, and fetching any necessary data.

Finally, the Razor Page executes its Razor view using the `PageModel` to generate the HTML response. The response returns back through the middleware pipeline and out to the user's browser.

How would this change if the request came from a client-side or mobile application? If you want to serve machine-readable JSON instead of HTML, what is different? As shown in figure 9.8, the answer is "very little." The main changes are related to switching from Razor Pages to controllers and actions, but as you saw in chapter 4, both approaches use the same general paradigms.

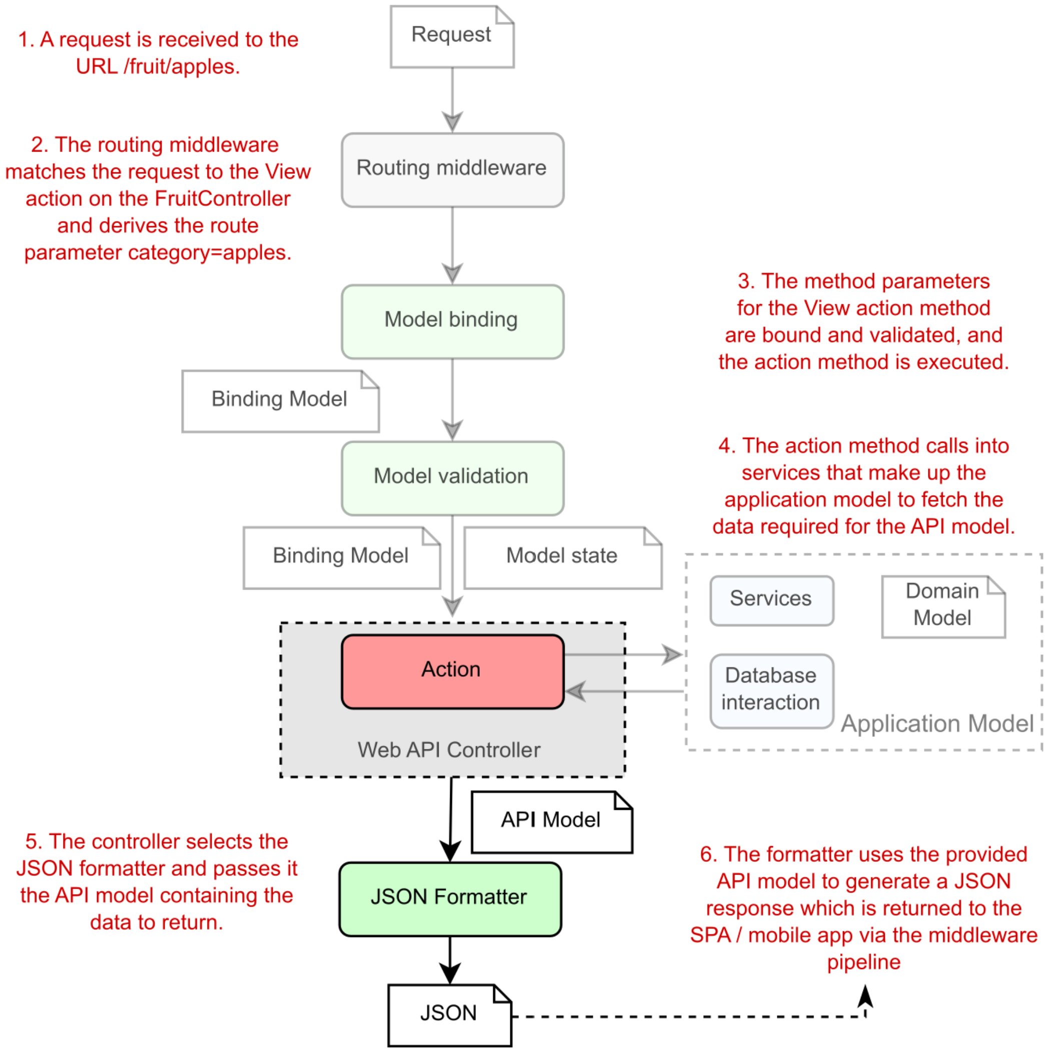

1. A request is received to the URL /fruit/apples.
Request
2. The routing middleware matches the request to the View action on the FruitController and derives the route parameter category=apples.
Routing middleware
3. The method parameters for the View action method are bound and validated, and the action method is executed.
Model binding
Binding Model
Model validation
4. The action method calls into services that make up the application model to fetch the data required for the API model.
Binding Model
Model state
Services
Domain Model
Database interaction
Application Model
Action
Web API Controller
API Model
5. The controller selects the JSON formatter and passes it the API model containing the data to return.
JSON Formatter
6. The formatter uses the provided API model to generate a JSON response which is returned to the SPA / mobile app via the middleware pipeline
JSON

Figure 9.8 A call to a Web API endpoint in an e-commerce ASP.NET Core web application. The ghosted portion of the diagram is identical to figure 9.7.

As before, the routing middleware selects an endpoint to invoke based on the incoming URL. For API Controllers this is a controller and action instead of a Razor Page.

After routing comes model binding, in which the binder creates a binding model and populates it with values from the request, exactly as for the Razor Pages request. Validation occurs in the same way, and the `ModelState` property on the `ControlerBase` base class is populated with any validation errors.

The action method is the equivalent of the Razor Page handler; it interacts with the application model in exactly the same way. This is an important point; by separating the behavior of your app into an application model, instead of incorporating it into your pages and controllers themselves, you're able to reuse the business logic of your application with multiple UI paradigms.

> **TIP** Where possible, keep your page handlers and controllers as simple as possible. Move all your "business logic" decisions into the services that make up your application model, and keep your Razor Pages and API controllers focused on the "mechanics" of interactive with a user.

After the application model has returned the data necessary to service the request—the fruit objects in the `apples` category—you see the first significant difference between API controllers and Razor Pages. Instead of adding values to the `PageModel` to be used in a Razor view, the action method creates an *API model*. This is analogous to the `PageModel`, but rather than containing data used to generate an HTML view, it contains the data that will be sent back in the response.

> **DEFINITION** ***View models*** **and** `PageModels` **contain both** ***data*** **required to build a response and** ***metadata*** **about** ***how*** **to build the response. API Models typically** ***only*** **contain the data to be returned in the response.**

When we looked at the Razor Pages app, we used the `PageModel` in conjunction with a Razor view template to build the final response. With the Web API app, we use the API model in conjunction with a *formatter*. A formatter, as the name suggests, serializes the API model into a machine-readable response, such as JSON or XML. The formatter forms the "V" in the Web API version of MVC, by choosing an appropriate representation for the data to return.

Finally, as for the Razor Pages app, the generated response is then sent back through the middleware pipeline, passing through each of the configured middleware components, and back to the original caller.

Hopefully, the parallels between Razor Pages and Web APIs are clear; the majority of behavior is identical, only the response varies. Everything from when the request arrives to the interaction with the application model is similar between the paradigms.

Most of the differences between Razor Pages and Web APIs are less to do with the way the framework works under the hood and are instead related to how the different paradigms are used. For example, in the next section, you'll learn how the routing constructs you learned about in chapter 5 are used with Web APIs, using attribute routing.

9.4 Attribute routing: linking action methods to URLs

In this section you'll learn about attribute routing: the mechanism for associating API controller actions with a given route template. You'll see how to associate controller actions with specific HTTP verbs like `GET` and `POST` and how to avoid duplication in your templates.

We covered route templates in depth in chapter 5 in the context of Razor Pages, and you'll be pleased to know that you use exactly the same route templates with API controllers. The only difference is how you *specify* the templates: with Razor Pages you use the `@model` directive, with API controllers you use routing attributes.

> **REMINDER** Both Razor Pages and API controllers use *attribute routing* under the hood. The alternative, *conventional* routing, is typically used with traditional MVC controllers and Views. As discussed previously, I don't recommend using that approach, so I don't cover conventional routing in this book.

With attribute routing, you decorate each action method in an API controller with an attribute, and provide the associated route template for the action method, as shown in the following listing.

Listing 9.4 Attribute routing example

```
public class HomeController: Controller
{
    [Route("")]                          #A
    public IActionResult Index()
    {
         /* method implementation*/
    }

    [Route("contact")]                   #B
    public IActionResult Contact()
    {
         /* method implementation*/
    }
}
```

#A The Index action will be executed when the "/" URL is requested.
#B The Contact action will be executed when the "/contact" URL is requested.

Each `[Route]` attribute defines a route template that should be associated with the action method. In the example provided, the `/` URL maps directly to the `Index` method and the `/contact` URL maps to the `Contact` method.

Attribute routing maps URLs to a specific action method, but a single action method can still have multiple route templates, and hence can correspond to multiple URLs. Each template must be declared with its own `RouteAttribute`, as shown in this listing, which shows the skeleton of a Web API for a car-racing game.

Listing 9.5 Attribute routing with multiple attributes

```
public class CarController
{
    [Route("car/start")]                    #A
    [Route("car/ignition")]                 #A
    [Route("start-car")]                    #A
    public IActionResult Start()            #B
    {
        /* method implementation*/
    }

    [Route("car/speed/{speed}")]            #C
    [Route("set-speed/{speed}")]            #C
    public IActionResult SetCarSpeed(int speed)
    {
        /* method implementation*/
    }
}
```

#A The Start method will be executed when any of these route templates are matched.
#B The name of the action method has no effect on the route template.
#C The RouteAttribute template can contain route parameters, in this case {speed}.

The listing shows two different action methods, both of which can be accessed from multiple URLs. For example, the `Start` method will be executed when any of the following URLS are requested:

- `/car/start`
- `/car/ignition`
- `/start-car`

These URLs are completely independent of the controller and action method names; only the value in the `RouteAttribute` matters.

> **NOTE By default, the controller and action name have no bearing on the URLs or route templates when `RouteAttribute`s are used.**

The templates used in route attributes are standard route templates, the same as you used in chapter 5. You can use literal segments and you're free to define route parameters that will extract values from the URL, as shown by the `SetCarSpeed` method in the previous listing. That method defines two route templates, both of which define a route parameter, `{speed}`.

> **TIP I've used multiple `[Route]` attributes on each action in this example, but it's best practice to expose your action at a *single* URL. This will make your API easier to understand and consume by other applications.**

Route parameters are handled in the very same way as for Razor Pages—they represent a segment of the URL that can vary. As for Razor Pages, the route parameters in your `RouteAttribute` templates can

- Be optional
- Have default values
- Use route constraints

For example, you could update the `SetCarSpeed` method in the previous listing to constrain `{speed}` to an integer and to default to `20` like so:

```
[Route("car/speed/{speed=20:int}")]
```

```
[Route("set-speed/{speed=20:int}")]
public IActionResult SetCarSpeed(int speed)
```

If you managed to get your head around routing in chapter 5, then routing with API controllers shouldn't hold any surprises for you. One thing you might begin noticing when you start using attribute routing is the amount you repeat yourself. Razor Pages removes a lot of repetition by using conventions to calculate route templates based on the Razor Page's filename.

Luckily, there are a few features available to make your life a little easier. In particular, combining route attributes and token replacement can help reduce duplication in your code.

9.4.1 Combining route attributes to keep your route templates DRY

Adding route attributes to all of your API controllers can get a bit tedious, especially if you're mostly following conventions where your routes have a standard prefix such as "api" or the controller name. Generally, you'll want to ensure you don't repeat yourself (DRY) when it comes to these strings. The following listing shows two action methods with a number of [Route] attributes. (This is for demonstration purposes only. Stick to one per action if you can!)

Listing 9.6 Duplication in RouteAttribute templates

```
public class CarController
{
    [Route("api/car/start")]                    #A
    [Route("api/car/ignition")]                 #A
    [Route("/start-car")]
    public IActionResult Start()
    {
        /* method implementation*/
    }

    [Route("api/car/speed/{speed}")]            #A
    [Route("/set-speed/{speed}")]
    public IActionResult SetCarSpeed(int speed)
    {
        /* method implementation*/
    }
}
```

#A Multiple route templates use the same "api/car" prefix

There's quite a lot of duplication here—you're adding "api/car" to most of your routes. Presumably, if you decided to change this to "api/vehicles", you'd have to go through each attribute and update it. Code like that is asking for a typo to creep in!

To alleviate this pain, it's possible to apply RouteAttributes to *controllers*, in addition to action methods, as you saw briefly in chapter 5. When a controller and an action method both have a route attribute, the overall route template for the method is calculated by combining the two templates.

Listing 9.7 Combining `RouteAttribute` templates

```
[Route("api/car")]
public class CarController
{
    [Route("start")]                              #A
    [Route("ignition")]                           #B
    [Route("/start-car")]                         #C
    public IActionResult Start()
    {
         /* method implementation*/
    }

    [Route("speed/{speed}")]                      #D
    [Route("/set-speed/{speed}")]                 #E
    public IActionResult SetCarSpeed(int speed)
    {
         /* method implementation*/
    }
}
```

#A Combines to give the "api/car/start" template
#B Combines to give the "api/car/ignition" template
#C Does not combine as starts with /, gives the "start-car" template
#D Combines to give the "api/car/speed/{speed}" template
#E Does not combine as starts with /, gives the "set-speed/{speed}" template

Combining attributes in this way can reduce some of the duplication in your route templates and makes it easier to add or change the prefixes (such as switching `"car"` to `"vehicle"`) for multiple action methods. To ignore the `RouteAttribute` on the controller and create an absolute route template, start your action method route template with a slash (`/`). Using a controller `RouteAttribute` reduces a lot of the duplication, but you can do one better by using token replacement.

9.4.2 Using token replacement to reduce duplication in attribute routing

The ability to combine attribute routes is handy, but you're still left with some duplication if you're prefixing your routes with the name of the controller, or if your route templates always use the action name. Luckily, you can simplify even further!

Attribute routes support the automatic replacement of the `[action]` and `[controller]` tokens in your attribute routes. These will be replaced with the name of the action and the controller (without the "Controller" suffix), respectively. The tokens are replaced after all attributes have been combined, so this is useful when you have controller inheritance hierarchies. This listing shows how you can create a `BaseController` class that you can use to apply a consistent route template prefix to *all* the API controllers in your application.

Listing 9.8 Token replacement in RouteAttributes

```
[Route("api/[controller]")]                        #A
public abstract class BaseController { }           #B
public class CarController : BaseController
{
    [Route("[action]")]                              #C
    [Route("ignition")]                              #D
    [Route("/start-car")]                            #E
    public IActionResult Start()
    {
         /* method implementation*/
    }
}
```

#A You can apply attributes to a base class, and derived classes will inherit them
#B Token replacement happens last, so [controller] is replaced with "car" not "base"
#C Combines and replaces tokens to give the "api/car/start" template
#D Combines and replaces tokens to give the "api/car/ignition" template
#E Does not combine with base attributes as it starts with /, so remains as "start-car"

When combined with everything you learned in chapter 5, we've covered pretty much everything there is to know about attribute routing now. There's just one more thing to consider: handling different HTTP request types like `GET` and `POST`.

9.4.3 Handling HTTP verbs with attribute routing

In Razor Pages, the HTTP verb, `GET` or `POST` for example, isn't part of the routing process. The `RoutingMiddleware` determines which Razor Page to execute based solely on the route template associated with the Razor Page. It's only when a Razor Page is about to be executed that the HTTP verb is used to decide which page handler to execute: `OnGet` for the `GET` verb, or `OnPost` for the `POST` verb for example.

With API controllers, things work a bit differently. For API controllers, the HTTP verb takes part in the routing process itself, so a `GET` request may be routed to one action, and a `POST` request may be routed to a different action, *even though the request used the same URL*. This pattern, where the HTTP verb is an important part of routing, is common in HTTP API design.

For example, imagine you're building an API to manage your calendar. You want to be able to list and create appointments. Well, a traditional HTTP REST service might define the following URLs and HTTP verbs to achieve this:

- `GET /appointments`—List all your appointments
- `POST /appointments`—Create a new appointment

Note that these two endpoints use the same URL, only the HTTP verb differs. The `[Route]` attribute we've used so far responds to *all* HTTP verbs which is no good for us here—we want to select a different action based on the combination or the URL *and* the verb. This pattern is common when building Web APIs and, luckily, it's easy to model in ASP.NET Core.

ASP.NET Core provides a set of attributes that you can use to indicate which verb an action should respond to. For example:

- `[HttpPost]` handles `POST` requests only
- `[HttpGet]` handles `GET` requests only

- [HttpPut] handles PUT requests only

There are similar attributes for all the standard HTTP verbs, like DELETE and OPTIONS. You can use these attributes instead of the [Route] attribute to specify that an action method should correspond to a single verb, as shown in the listing below.

Listing 9.9 Using HTTP verb attributes with attribute routing

```
public class AppointmentController
{
    [HttpGet("/appointments")]                #A
    public IActionResult ListAppointments()   #A
    {                                         #A
        /* method implementation */           #A
    }                                         #A

    [HttpPost("/appointments")]               #B
    public IActionResult CreateAppointment()  #B
    {                                         #B
        /* method implementation */           #B
    }                                         #B
}
```

#A Only executed in response to GET /appointments
#B Only executed in response to POST /appointments

If your application receives a request that matches the route template of an action method, but which *doesn't* match the required HTTP verb, you'll get a 405 Method not allowed error response. For example, if you send a DELETE request to the /appointments URL in the previous listing, you'll get a 405 error response.

Attribute routing has been used with API controllers since the first days of ASP.NET Core, as it allows tight control over the URLs that your application exposes. When you're building API controllers, there is some code that you find yourself writing repeatedly. The [ApiController] attribute, introduced in ASP.NET Core 2.1 is designed to handle some of these for you, and reduce the amount of boilerplate you need.

9.5 Using common conventions with ApiControllerAttribute

In this section you'll learn about the [ApiController] attribute and how it can reduce the amount of code you need to write to create consistent Web API controllers. You'll learn about the conventions it applies, why they're useful, and how to turn them off if you need to.

The [ApiController] attribute was introduced in .NET Core 2.1 to simplify the process of creating Web API controllers. To understand what it does, it's useful to look at an example of how you might write a Web API controller *without* the [ApiController] attribute, and compare that to the code required to achieve the same thing with the attribute.

Listing 9.10 Creating a Web API controller without the [ApiController] attribute

```
public class FruitController : ControllerBase
```

```
{
    List<string> _fruit = new List<string>                           #A
    {                                                                #A
        "Pear", "Lemon", "Peach"                                     #A
    };                                                               #A

    [HttpPost("fruit")]                                              #B
    public ActionResult Update([FromBody] UpdateModel model)         #C
    {
        if (!ModelState.IsValid)                                     #D
        {                                                            #D
             return BadRequest(                                      #D
                new ValidationProblemDetails(ModelState));           #D
        }                                                            #D

        if (model.Id < 0 || model.Id > _fruit.Count)
        {
            return NotFound(new ProblemDetails()                     #E
            {                                                        #E
                Status = 404,                                        #E
                Title = "Not Found",                                 #E
                Type = "https://tools.ietf.org/html/rfc7231"         #E
                       + "#section-6.5.4",                           #E
            });                                                      #E
        }                                                            #E

        _fruit[model.Id] = model.Name;                               #F
        return Ok();                                                 #F
    }

    public class UpdateModel
    {
        public int Id { get; set; }

        [Required]                                                   #G
        public string Name { get; set; }                             #G
    }
}
```

#A The list of strings serves as the application model in this example.
#B Web APIs use attribute routing to define the route templates
#C The [FromBody] attribute indicates the parameter should be bound to the request body
#D You need to check if model validation succeeded, and return a 400 response if it failed
#E If the data sent does not contain a valid ID, return a 404 ProblemDetails response
#F Update the model, and return a 200 Response
#G The UpdateModel is only valid if the Name value is provided, as set by the [Required] attribute

This example demonstrates many common features and patterns used with Web API controllers:

- Web API controllers read data from the body of a request, typically sent as JSON. To ensure the body is read as JSON and not as form values, you have to apply the `[FromBody]` attribute to the method parameters to ensure it is model bound correctly.
- As discussed in chapter 6, after model binding, the model is validated, but it's up to you to act on the validation results. You should return a `400 Bad Request` response if

the values provided failed validation. You typically want to provide details of *why* the request was invalid: this is done in listing 9.10 by returning a `ValidationProblemDetails` object, built from the `ModelState`.

- Whenever you return an error status, such as a `404 Not Found`, where possible you should return details of the problem that allow the caller to diagnose the issue. The `ProblemDetails` class is the recommended way of doing that in ASP.NET Core.

The code in listing 9.10 is representative of what you might see in an ASP.NET Core API controller, *prior* to .NET Core 2.1. The introduction of the `[ApiController]` attribute in .NET Core 2.1 (and subsequent refinement in later versions), makes this *same* code much simpler, as shown in the following listing:

Listing 9.11 Creating a Web API controller with the [ApiController] attribute

```
[ApiController]                                        #A
public class FruitController : ControllerBase
{
    List<string> _fruit = new List<string>
    {
        "Pear", "Lemon", "Peach"
    };

    [HttpPost("fruit")]
    public ActionResult Update(UpdateModel model)      #B
    {                                                  #C
        if (model.Id < 0 || model.Id > _fruit.Count)
        {
            return NotFound();                         #D
        }

        _fruit[model.Id] = model.Name;

        return Ok();
    }

    public class UpdateModel
    {
        public int Id { get; set; }

        [Required]
        public string Name { get; set; }
    }
}
```

#A Adding the [ApiController] attribute applies several conventions common to API controllers
#B The [FromBody] attribute is assumed for complex action parameter arguments
#C The model validation is automatically checked, and if invalid, returns a 400 response
#D Error status codes are automatically converted to a ProblemDetails object

If you compare listing 9.10 to listing 9.11, you'll see that all the highlighted code can be removed and replaced with the `[ApiController]` attribute. The `[ApiController]` attribute automatically applies several conventions to your controllers:

- *Attribute routing*. You must use attribute routing with your controllers, you can't use conventional routing. Not that you would, as we've only discussed this approach for API controllers anyway!
- *Automatic 400 responses*. I said in chapter 6 that you should *always* check the value of `ModelState.IsValid` in your Razor Page handlers and MVC actions, but the `[ApiController]` attribute does this for you, by adding a *filter*. We'll cover filters in detail in chapter 13.
- *Model binding source inference*. Without the `[ApiController]` attribute, complex types are assumed to be passed as *form* values in the request body. For Web APIs, it's much more common to pass data as JSON, which ordinarily requires adding the `[FromBody]` attribute. The `[ApiController]` attribute takes care of that for you.
- `ProblemDetails` *for error codes*. You often want to return a consistent set of data when an error occurs in your API. `ProblemDetails` is a type based on a web standard that serves as this consistent data. The `[ApiController]` attribute will intercept any error status codes returned by your controller (for example, a 404 Not Found response), and convert them into the `ProblemDetails` type automatically.

A key feature of the `[ApiController]` attribute is using the *Problem Details*[40] format to return errors in a consistent format across all your controllers. A typical `ProblemDetails` object looks something like the following, for example when a `ValidationProblemDetails` object is generated automatically for an invalid request:

```
{
  type: "https://tools.ietf.org/html/rfc7231#section-6.5.1"
  title: "One or more validation errors occurred."
  status: 400
  traceId: "|17a2706d-43736ad54bed2e65."
  errors: {
    name: ["The name field is required."]
  }
}
```

The `[ApiController]` conventions can significantly reduce the amount of boilerplate code you have to write. They also ensure consistency across your whole application. For example, you can be sure that all your controllers will return the same error type, `ValidationProblemDetails` (a sub-type of `ProblemDetails`) when a bad request is received.

Converting all your errors to ProblemDetails

The `[ApiController]` attribute ensures that all error responses returned by your API controllers are converted into `ProblemDetails` objects, which keeps the error responses consistent across your application.

[40] Problem Details is a proposed standard for handling errors in a machine-readable way that is gaining popularity. You can find the specification here, but be warned, it makes for dry reading https://tools.ietf.org/html/rfc7807.

The only problem with this is that your API controllers aren't the *only* thing that could generate errors. For example, if a URL is received that doesn't match any action in your controllers, the "end-of-the-pipeline" middleware we discussed in chapter 3 would generate a 404 Not Found response. As this error is generated *outside* of the API controllers, it won't use `ProblemDetails`. Similarly, when your code throws an exception, you want this to be returned as a `ProblemDetails` object too, but this doesn't happen by default.

In chapter 3 I described several types of error handling middleware that you could use to handle these cases, but it can be complicated to handle all the edge cases. I prefer to use a community-created package, *Hellang.Middleware.ProblemDetails*, which takes care of this for you. You can read about how to use this package at https://andrewlock.net/handling-web-api-exceptions-with-problemdetails-middleware/.

As is common in ASP.NET Core, you will be most productive if you follow the conventions rather than trying to fight them. However, if you don't like some of the conventions, or want to customize them, then you can easily do so.

You can customize the conventions your application uses by calling `ConfigureApiBehaviorOptions()` on the `IMvcBuilder` object returned from the `AddControllers()` method in your Startup.cs file. For example, you could disable the automatic `400` responses on validation failure, as shown in the listing below.

Listing 9.12 Customizing [ApiAttribute] behaviors

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers()
            .ConfigureApiBehaviorOptions(options =>                #A
            {
                options.SuppressModelStateInvalidFilter = true;    #B
            })
    }

    // ...
}
```

#A Control which conventions are applied by providing a configuration lambda
#B This would disable the automatic 400 responses for invalid requests

The ability to customize each aspect of ASP.NET Core is one of the features that sets it apart from the previous version of ASP.NET. ASP.NET Core configures the vast majority of its internal components using one of two mechanisms—dependency injection or by configuring an `Options` object when you add the service to your application, as you'll see in chapters 10 (dependency injection) and 11 (`Options` object).

In the next section you'll learn how to control the format of the data returned by your Web API controllers—whether that's JSON, XML, or a different, custom format.

9.6 Generating a response from a model

This brings us to the final section in this chapter: formatting a response. It’s common for API controllers to return JSON these days, but that’s not *always* the case. In this section you’ll learn about content negotiation and how to enable additional output formats such as XML. You’ll also learn about an important change in the JSON formatter for ASP.NET Core 3.0.

Consider this scenario: You’ve created a Web API action method for returning a list of cars, as in the following listing. It invokes a method on your application model, which hands back the list of data to the controller. Now you need to format the response and return it to the caller.

Listing 9.13 A Web API controller to return a list of cars

```
[ApiController]
public class CarsController : Controller
{
    [HttpGet("api/cars")]                             #A
    public IEnumerable<string> ListCars()             #B
    {
        return new string[]                           #C
            { "Nissan Micra", "Ford Focus" };         #C
    }
}
```

#A The action is executed with a request to GET /api/cars.
#B The API Model containing the data is an IEnumerable<string>.
#C This data would normally be fetched from the application model.

You saw in section 9.2 that it’s possible to return data directly from an action method, in which case, the middleware formats it and returns the formatted data to the caller. But how does the middleware know which format to use? After all, you could serialize it as JSON, as XML, even with a simple `ToString()` call.

The process of determining the format of data to send to clients is known generally as *content negotiation* (conneg). At a high level, the client sends a header indicating the types of content it can understand—the `Accept` header—and the server picks one of these, formats the response, and sends a `content-type` header in the response, indicating which it chose.

The accept and content-type headers

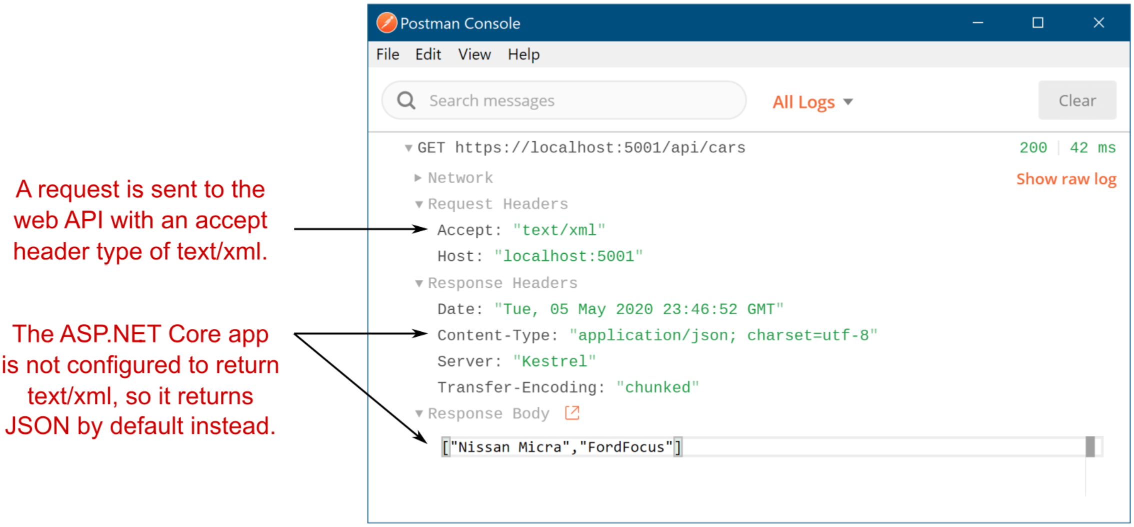
The `accept` header is sent by a client as part of a request to indicate the type of content that the client can handle. It consists of a number of MIME types,[a] with optional weightings (from 0 to 1) to indicate which type would be preferred. For example, the `application/json,text/xml;q=0.9,text/plain;q=0.6` header indicates that the client can accept JSON, XML, and plain text, with weightings of 1.0, 0.9, and 0.6, respectively. JSON has a weighting of 1.0, as no explicit weighting was provided. The weightings can be used during content negotiation to choose an optimal representation for both parties.

The `content-type` header describes the data sent in a request or response. It contains the MIME type of the data, with an optional character encoding. For example, the `application/json; charset=utf-8` header would indicate that the body of the request or response is JSON, encoded using UTF-8.

[a] For more on MIME types see https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types.

You're not forced into *only* sending a `content-type` the client expects and, in some cases, you may not even be *able* to handle the types it requests. What if a request stipulates it can only accept Excel spreadsheets? It's unlikely you'd support that, even if that's the only `content-type` the request contains!

When you return an API model from an action method, whether directly (as in the listing 9.13) or via an `OkResult` or other `StatusCodeResult`, ASP.NET Core will always return *something*. If it can't honor any of the types stipulated in the `Accept` header, it will fall back to returning JSON by default. Figure 9.9 shows that even though XML was requested, the API controller formatted the response as JSON.



```
▼ GET https://localhost:5001/api/cars                              200 | 42 ms
  ▸ Network
  ▼ Request Headers
     Accept: "text/xml"
     Host: "localhost:5001"
  ▼ Response Headers
     Date: "Tue, 05 May 2020 23:46:52 GMT"
     Content-Type: "application/json; charset=utf-8"
     Server: "Kestrel"
     Transfer-Encoding: "chunked"
  ▼ Response Body
     ["Nissan Micra","FordFocus"]
```

A request is sent to the web API with an accept header type of text/xml.

The ASP.NET Core app is not configured to return text/xml, so it returns JSON by default instead.

Figure 9.9 Even though the request was made with an `Accept` header of `text/xml`, the response returned was JSON, as the server was not configured to return XML.

> **NOTE** In the previous version of ASP.NET, objects were serialized to JSON using PascalCase, where properties start with a capital letter. In ASP.NET Core, objects are serialized using camelCase by default, where properties start with a lowercase letter.

However, the data is sent, it's serialized by an `IOutputFormatter` implementation. ASP.NET Core ships with a limited number of output formatters out of the box, but as always, it's easy to add additional ones, or change the way the defaults work.

9.6.1 Customizing the default formatters: adding XML support

As with most of ASP.NET Core, the Web API formatters are completely customizable. By default, only formatters for plain text (`text/plain`), HTML (`text/html`), and JSON (`application/json`) are configured. Given the common use-case of SPAs and mobile applications, this will get you a long way. But sometimes you need to be able to return data in a different format, such as XML.

Newtonsoft.Json vs System.Text.Json

Newtonsoft.Json, also known as Json.NET, has for a long time been the canonical way to work with JSON in .NET. It’s compatible with every version of .NET under the sun, and will no doubt be familiar to virtually all .NET developers. Its reach was so great, that even ASP.NET Core took a dependency on it!

In ASP.NET Core 3.0, that all changed with the introduction of System.Text.Json. This is a new library which focuses on performance and is built in to .NET Core 3.0. In ASP.NET Core 3.0 onwards, ASP.NET Core uses System.Text.Json by default instead of Newronsoft.Json.

The main difference between the libraries is that System.Text.Json is very picky about its JSON. It will generally only deserialize JSON that matches its expectations. For example, System.Text.Json won’t deserialize JSON that uses single quotes around strings; you have to use double quotes.

If you’re creating a new application, this is generally not a problem—you quickly learn to generate the correct JSON! But if you’re migrating an application from ASP.NET Core 2.0, or are receiving JSON from a third-party, these limitations can be real stumbling blocks.

Luckily, you can easily switch back to the Newtonsoft.Json library instead. Install the *Microsoft.AspNetCore.Mvc.NewtonsoftJson* package into your project, and update the `AddControllers()` method in Startup.cs to the following:

```
services.AddControllers()
    .AddNewtonsoftJson();
```

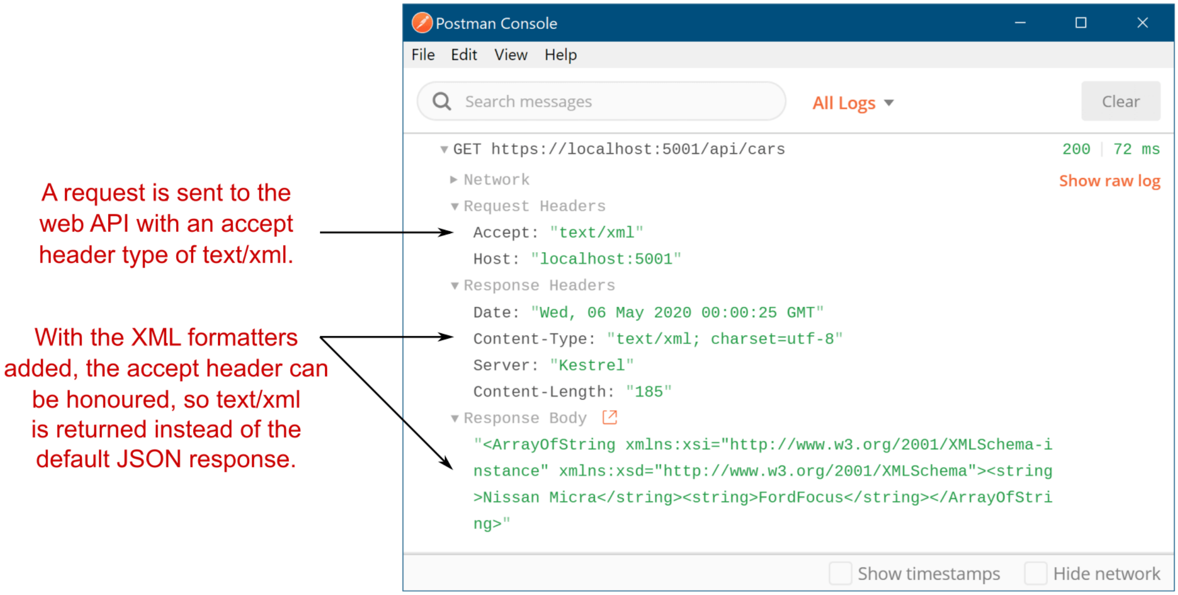
This will switch ASP.NET Core’s formatters to use Newtonsoft.Json behind the scenes, instead of System.Text.Json. For more details on the differences between the libraries, see https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-text-json-migrate-from-newtonsoft-how-to. For more advice on when to switch to the Newtonsoft.Json formatter, see https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting#add-newtonsoftjson-based-json-format-support.

You can add XML output to your application by adding an *output formatter*. You configure your application’s formatters in Startup.cs, by customizing the `IMvcBuilder` object returned from `AddControllers()`. To add the XML output formatter[41], use the following:

[41] Technically this also adds an XML *input formatter* as well, which means your application can now *receive* XML in requests too. For a detailed look at formatters, including creating a custom formatter, see the documentation at https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/custom-formatters.

```
services.AddControllers()
    .AddXmlSerializerFormatters();
```

With this simple change, your API controllers can now format responses as XML. Running the same request as shown in figure 9.9 with XML support enabled means the app will respect the `text/xml accept` header. The formatter serializes the `string` array to XML as requested instead of defaulting to JSON, as shown in figure 9.10.



```
▾ GET https://localhost:5001/api/cars                         200 | 72 ms
  ▸ Network
  ▾ Request Headers
    Accept: "text/xml"
    Host: "localhost:5001"
  ▾ Response Headers
    Date: "Wed, 06 May 2020 00:00:25 GMT"
    Content-Type: "text/xml; charset=utf-8"
    Server: "Kestrel"
    Content-Length: "185"
  ▾ Response Body
    "<ArrayOfString xmlns:xsi="http://www.w3.org/2001/XMLSchema-i
    nstance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"><string
    >Nissan Micra</string><string>FordFocus</string></ArrayOfStri
    ng>"
```

A request is sent to the web API with an accept header type of text/xml.

With the XML formatters added, the accept header can be honoured, so text/xml is returned instead of the default JSON response.

Figure 9.10 With the XML output formatters added, the `text/xml Accept` header is respected and the response can be serialized to XML.

This is an example of content negotiation, where the client has specified what formats it can handle and the server selects one of those, based on what it can produce. This approach is part of the HTTP protocol, but there are some quirks to be aware of when relying on it in ASP.NET Core. You won't often run into these, but if you're not aware of them when they hit you, they could have you scratching your head for hours!

9.6.2 Choosing a response format with content negotiation

Content negotiation is where a client says which types of data it can accept using the `accept` header and the server picks the best one it can handle. Generally speaking, this works as you'd hope: the server formats the data using a type the client can understand.

The ASP.NET Core implementation has some special cases that are worth bearing in mind:

- By default, the middleware will only return `application/json`, `text/plain` and `text/html` MIME types. You can add additional `IOutputFormatters` to make other types available, as you saw in the previous section for `text/xml`.
- By default, if you return `null` as your API model, whether from an action method, or by passing `null` in a `StatusCodeResult`, the middleware will return a `204 No Content` response.
- When you return a `string` as your API model, if no `Accept` header is set, the middleware will format the response as `text/plain`.
- When you use any other class as your API model and there's either no `Accept` header or none of the supported formats were requested, the first formatter that can generate a response will be used (typically JSON by default).
- If the middleware detects that the request is probably from a browser (the `accept` header contains `*/*`), then it will *not* use conneg. Instead, it will format the response as though no `accept` header was provided, using the default formatter (typically JSON).

These defaults are relatively sane, but they can certainly bite you if you're not aware of them. The last point in particular, where the response to a request from a browser is virtually always formatted as JSON has certainly caught me out when trying to test XML requests locally!

As you should expect by now, all of these rules are configurable; you can easily change the default behavior in your application if it doesn't fit your requirements. For example, the following listing, taken from Startup.cs, shows how you can force the middleware to respect the browser's `Accept` header, and remove the `text/plain` formatter for `string`s.

Listing 9.14 Customizing MVC to respect the browser `Accept` header in Web APIs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(options =>                          #A
    {
        options.RespectBrowserAcceptHeader = true;                  #B
         options.OutputFormatters.RemoveType<StringOutputFormatter>();    #C
    });
}
```

#A AddControllers has an overload that takes a lambda function
#B False by default, a number of other properties are also available to be set
#C Removes the output formatter that formats strings as text/plain

In most cases, conneg should work well for you out of the box, whether you're building an SPA or a mobile application. In some cases, you may find you need to bypass the usual conneg mechanisms for specific action methods, and there are a number of ways to achieve this, but I won't cover them in this book as I've found I rarely need to use them. For details, see the documentation at https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting.

That brings us to the end of this chapter on Web APIs and, with it, part 1 of this book! It's been a pretty intense tour of ASP.NET Core, with a heavy focus on Razor Pages and the MVC pattern. By making it this far, you now have all the knowledge you need to start building simple applications using Razor Pages, or to create a Web API server for your SPA or mobile app.

In part 2, you'll get into some juicy topics, in which you'll learn the details needed to build complete apps, like adding users to your application, saving data to a database, and how to deploy your application.

In chapter 10, we'll look at dependency injection in ASP.NET Core and how it helps create loosely coupled applications. You'll learn how to register the ASP.NET Core framework services with a container and set up your own classes as dependency-injected services. Finally, you'll see how to replace the built-in container with a third-party alternative.

9.7 Summary

- A Web API exposes a number of methods or endpoints that can be used to access or change data on a server. It's typically accessed using HTTP by mobile or client-side web applications.
- Web API action methods can return data directly or can use `ActionResult<T>` to generate an arbitrary response.
- If you return more than one type of result from an action method, the method signature must return `ActionResult<T>`.
- Web APIs follow the same MVC design pattern as traditional web applications. The formatters which generate the final response form the view.
- The data returned by a Web API action is called an API model. It contains the data the middleware will serialize and send back to the client. This differs from view models and `PageModel`s which contain both data and metadata about how to generate the response.
- Web APIs are associated with route templates by applying `RouteAttribute`s to your action methods. These give you complete control over the URLs that make up your application's API.
- Route attributes applied to a controller combine with attributes on action methods to form the final template. These are also combined with attributes on inherited base classes. You can use inherited attributes to reduce the amount of duplication in the attributes where you're using a common prefix on your routes, for example.
- By default, the controller and action name have no bearing on the URLs or route templates when you use attribute routing. However, you can use the `"[controller]"` and `"[action]"` tokens in your route templates to reduce repetition. They'll be replaced with the current controller and action name.
- The `[HttpPost]` and `[HttpGet]` attributes allow choosing between actions based on the request's HTTP verb when two actions correspond to the same URL. This is a common pattern in RESTful applications.

- The `[ApiController]` attribute applies several common conventions to your controllers. Controllers decorated with the attribute will automatically bind to a request's body instead of using form values, automatically generates a `400 Bad Request` response for invalid requests, and will return `ProblemDetails` objects for status code errors. This can dramatically reduce the amount of boilerplate code you must write.
- You can control which of the conventions to apply by using the `ConfigureApiBehaviorOptions()` method and providing a configuration lambda. This is useful if you need to fit your API to an existing specification for example.
- By default, ASP.NET Core formats the API model returned from a Web API controller as JSON. Virtually every platform can handle JSON, making your API highly interoperable.
- In contrast to the previous version of ASP.NET, JSON data is serialized using camelCase rather than PascalCase. You should consider this change if you get errors or missing values when migrating from ASP.NET to ASP.NET Core.
- ASP.NET Core 3.0 uses System.Text.Json for JSON serialization and deserialization, which is a high performance, strict library that is part of .NET Core 3.0. You can replace this serializer with the common Newtonsoft.Json formatter, by calling `AddNewtonsoftJson()`, on the return value from `services.AddControllers()`.
- Content negotiation occurs when the client specifies the type of data it can handle and the server chooses a return format based on this. It allows multiple clients to call your API and receive data in a format they can understand.
- By default, ASP.NET Core can return `text/plain`, `text/html`, and `application/ json`, but you can add additional formatters if you need to support other formats.
- You can add XML formatters by calling `AddXmlSerializerFormatters()` on the return value from `services.AddControllers()` in your Startup class. These let you format the response as XML, as well as receive XML in a request body.
- Content negotiation isn't used when the accept header contains `*/*`, such as in most browsers. Instead, your application will use the default formatter, JSON. You can disable this option by modifying the `RespectBrowserAcceptHeader` option when adding your MVC controller services in Startup.cs.

Part 2

Building complete applications

We covered a lot of ground in part 1. You saw how an ASP.NET Core application is composed of middleware and we focused heavily on the Razor Pages framework. You saw how to use it to build traditional server-side-rendered apps using Razor syntax and how to build APIs for mobile and client-side apps.

In part 2, we dive deeper into the framework and look at a variety of components that you'll inevitably need when you want to build more complex apps. By the end of this part, you'll be able to build dynamic applications, customized to specific users, that can be deployed to multiple environments, each with a different configuration.

ASP.NET Core uses dependency injection (DI) throughout its libraries, so it's important that you understand how this design pattern works. In chapter 10, I introduce DI, why it is used, and how to configure the services in your applications to use DI.

Chapter 11 looks at the ASP.NET Core configuration system, which lets you pass configuration values to your app from a range of sources—JSON files, environment variables, and many more. You'll learn how to configure your app to use different values depending on the environment in which it is running, and how to bind strongly typed objects to your configuration to help reduce runtime errors.

Most web applications require some sort of data storage, so in chapter 12, I introduce Entity Framework Core (EF Core). This is a new, cross-platform library that makes it easier to connect your app to a database. EF Core is worthy of a book in and of itself, so I only provide a brief introduction. I show you how to create a database and how to insert, update, and query simple data.

In chapters 13 through 15, we look at how to build more complex applications. You'll see how you can add ASP.NET Core Identity to your apps so that users can log in and enjoy a customized experience. You'll learn how to protect your app using authorization to ensure only certain users can access certain action methods, and you'll see how to refactor your app to extract common code out of your Razor Pages and API controllers using filters.

In the final chapter of this part, I cover the steps required to make an app live, including how to publish your app to IIS, how to configure the URLs your app listens on, and how to optimize your client-side assets for improved performance.

10

Service configuration with dependency injection

This chapter covers

- **Understanding the benefits of dependency injection**
- **How ASP.NET Core uses dependency injection**
- **Configuring your services to work with dependency injection**
- **Choosing the correct lifetime for your services**

In part 1 of this book, you saw the bare bones of how to build applications with ASP.NET Core. You learned how to compose middleware to create your application and how to use the MVC pattern to build traditional web applications with Razor Pages and Web APIs. This gave you the tools to start building simple applications.

In this chapter, you'll see how to use *dependency injection* (DI) in your ASP.NET Core applications. DI is a design pattern that helps you develop loosely coupled code. ASP.NET Core uses the pattern extensively, both internally in the framework and in the applications you build, so you'll need to use it in all but the most trivial of applications.

You may have heard of DI before, and possibly even used it in your own applications. If so, this chapter shouldn't hold many surprises for you. If you haven't used DI before, never fear, I'll make sure you're up to speed by the time the chapter is done!

This chapter starts by introducing DI in general, the principles it drives, and why you should care about it. You'll see how ASP.NET Core has embraced DI throughout its implementation and why you should do the same when writing your own applications.

Once you have a solid understanding of the concept, you'll see how to apply DI to your own classes. You'll learn how to configure your app so that the ASP.NET Core framework can create your classes for you, removing the pain of having to create new objects manually in your code. Toward the end of the chapter, you'll learn how to control how long your objects are used for and some of the pitfalls to be aware of when you come to write your own applications.

In chapter 19, we'll revisit some of the more advanced ways to use DI, including how to wire up a third-party DI container. For now though, let's get back to basics: what is DI and why should you care about it?

10.1 Introduction to dependency injection

This section aims to give you a basic understanding of what dependency injection is, why you should care about it, and how ASP.NET Core uses it. The topic itself extends far beyond the reach of this single chapter. If you want a deeper background, I highly recommend checking out Martin Fowler's articles online.[42]

> **TIP** For a more directly applicable read with many examples in C#, I recommend picking up *Dependency Injection Principles, Practices, and Patterns* by Steven van Deursen and Mark Seemann (Manning, 2019).

The ASP.NET Core framework has been designed from the ground up to be modular and to adhere to "good" software engineering practices. As with anything in software, what is considered best practice varies over time, but for object-oriented programming, the SOLID[43] principles have held up well.

On that basis, ASP.NET Core has *dependency injection* (sometimes called *dependency inversion*, *DI*, or *inversion of control*[44]) baked into the heart of the framework. Whether or not you want to use it within your own application code, the framework libraries themselves depend on it as a concept.

I begin this section by starting with a common scenario: a class in your application depends on a different class, which in turn depends on another. You'll see how dependency injection can help alleviate this chaining of dependencies for you and provide a number of extra benefits.

10.1.1 Understanding the benefits of dependency injection

When you first started programming, the chances are you didn't immediately use a DI framework. That's not surprising, or even a bad thing; DI adds a certain amount of extra

[42] Martin Fowler's website at https://martinfowler.com is a gold mine of best-practice goodness. One of the most applicable articles to this chapter can be found at www.martinfowler.com/articles/injection.html.

[43] SOLID is a mnemonic for Single responsibility, Open-closed, Liskov substitution, Interface segregation, and Dependency inversion: https://en.wikipedia.org/wiki/SOLID_(object-oriented_design).

[44] Although related, dependency injection and dependency inversion are two different things. I cover both in a general sense in this chapter, but for a good explanation of the differences, see this post by Derick Bailey: https://lostechies.com/derickbailey/2011/09/22/dependency-injection-is-not-the-same-as-the-dependency-inversion-principle/.

wiring that's often not warranted in simple applications or when you're getting started. But when things start to get more complex, DI comes into its own as a great tool to help keep that complexity in check.

Let's consider a simple example, written without any sort of DI. Imagine a user has registered on your web app and you want to send them an email. This listing shows how you might approach this initially in an API controller.

> **NOTE** I'm using an API controller for this example, but I could just as easily have used a Razor Page. Razor Pages and API controllers both use constructor dependency injection, as you'll see in section 10.2.

Listing 10.1 Sending an email without DI when there are no dependencies

```
public class UserController : ControllerBase
{
    [HttpPost("register")]
    public IActionResult RegisterUser(string username)      #A
    {
        var emailSender = new EmailSender();            #B
        emailSender.SendEmail(username);                #C
        return Ok();
    }
}
```

#A The action method is called when a new user is created
#B Creates a new instance of EmailSender
#C Uses the new instance to send the email

In this example, the `RegisterUser` action on `UserController` executes when a new user registers on your app. This creates a new instance of an `EmailSender` class, and calls `SendEmail()` to send the email. The `EmailSender` class is the one that does the sending of the email. For the purposes of this example, you can imagine it looks something like this:

```
public class EmailSender
{
    public void SendEmail(string username)
    {
        Console.WriteLine($"Email sent to {username}!");
    }
}
```

`Console.Writeline` "stands in" for the real process of sending the email.

> **NOTE** Although I'm using sending email as a simple example, in practice you might want to move this code out of your Razor Page and controller classes entirely. This type of asynchronous task is well suited to using message queues and a background process. For more details, see https://docs.microsoft.com/dotnet/architecture/microservices/.

If the `EmailSender` class is as simple as the previous example and it has no dependencies, then you might not see any need to adopt a different approach to creating objects. And to an extent, you'd be right. But what if you later update your implementation of `EmailSender` so that it doesn't implement the whole email-sending logic itself?

In practice, `EmailSender` would need to do many things to send an email. It would need to

- Create an email message
- Configure the settings of the email server
- Send the email to the email server

Doing all of that in one class would go against the single responsibility principle (SRP), so you'd likely end up with `EmailSender` depending on other services. Figure 10.1 shows how this web of dependencies might look. `UserController` wants to send an email using `EmailSender`, but to do so, it also needs to create the `MessageFactory`, `NetworkClient`, and `EmailServerSettings` objects that `EmailSender` depends on.

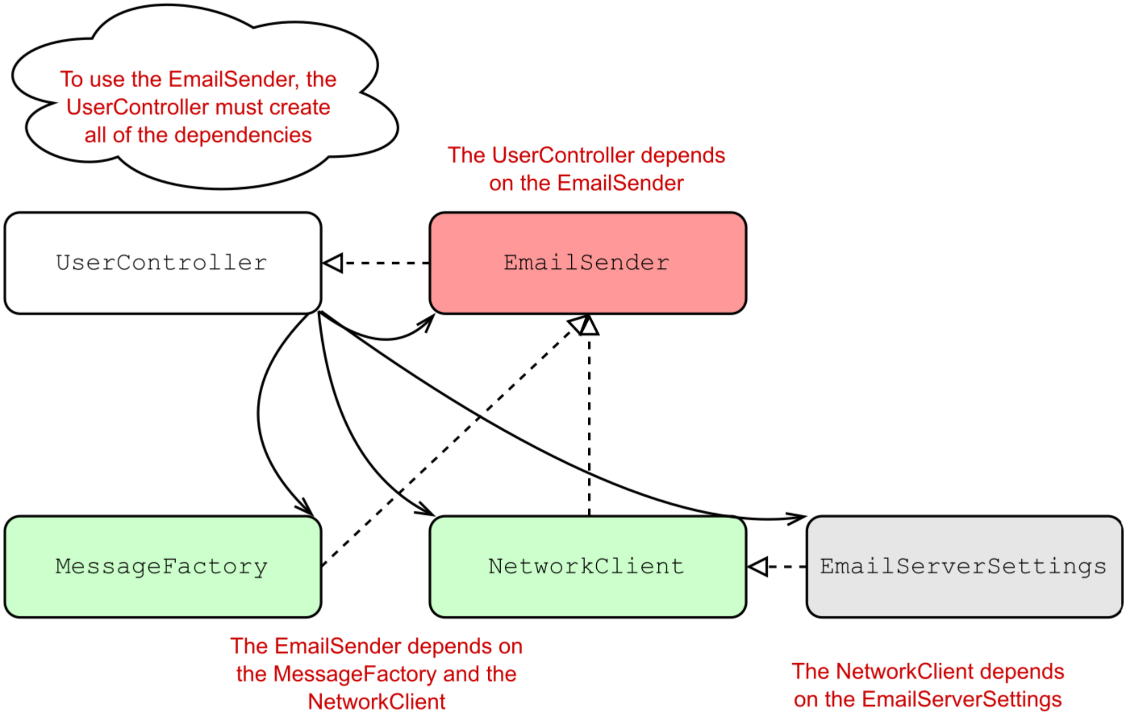

To use the EmailSender, the UserController must create all of the dependencies
The UserController depends on the EmailSender
UserController
EmailSender
MessageFactory
NetworkClient
EmailServerSettings
The EmailSender depends on the MessageFactory and the NetworkClient
The NetworkClient depends on the EmailServerSettings

Figure 10.1 Dependency diagram without dependency injection. `UserController` **indirectly depends on all the other classes; so, it has to create them all.**

Each class has a number of dependencies, so the "root" class, in this case `UserController`, needs to know how to create every class it depends on, as well as every class its *dependencies* depend on. This is sometimes called the *dependency graph*.

> **DEFINITION** The *dependency graph* is the set of objects that must be created in order to create a specific requested "root" object.

`EmailSender` depends on the `MessageFactory` and `NetworkClient` objects, so they're provided via the constructor, as shown here.

Listing 10.2 A service with multiple dependencies

```
public class EmailSender
{
    private readonly NetworkClient _client;                         #A
    private readonly MessageFactory _factory;                       #A
    public EmailSender(MessageFactory factory, NetworkClient client)  #B
    {                                                               #B
        _factory = factory;                                         #B
        _client = client;                                           #B
    }                                                               #B
    public void SendEmail(string username)
    {
        var email = _factory.Create(username);                      #C
        _client.SendEmail(email);                                   #C
        Console.WriteLine($"Email sent to {username}!");
    }
}
```

#A The EmailSender now depends on two other classes.
#B Instances of the dependencies are provided in the constructor.
#C The EmailSender coordinates the dependencies to create and send an email.

On top of that, the `NetworkClient` class that `EmailSender` depends on also has a dependency on an `EmailServerSettings` object:

```
public class NetworkClient
{
    private readonly EmailServerSettings _settings;
    public NetworkClient(EmailServerSettings settings)
    {
        _settings = settings;
    }
}
```

This might feel a little contrived, but it's common to find this sort of chain of dependencies. In fact, if you *don't* have this in your code, it's probably a sign that your classes are too big and aren't following the single responsibility principle.

So, how does this affect the code in `UserController`? The following listing shows how you now have to send an email, if you stick to `new`-ing up objects in the controller.

Listing 10.3 Sending email without DI when you manually create dependencies

```
public IActionResult RegisterUser(string username)
{
    var emailSender = new EmailSender(                         #A
        new MessageFactory(),                                  #B
        new NetworkClient(                        #C
            new EmailServerSettings               #D
            (                                     #D
                host: "smtp.server.com",          #D
                port: 25                          #D
            ))                                    #D
        );
    emailSender.SendEmail(username);              #E
    return Ok();
}
```

#A To create EmailSender, you must create all of its dependencies.
#B You need a new MessageFactory.
#C The NetworkClient also has dependencies.
#D You're already two layers deep, but there could feasibly be more.
#E Finally, you can send the email.

This is turning into some gnarly code. Improving the design of `EmailSender` to separate out the different responsibilities has made calling it from `UserController` a real chore. This code has several issues, among them:

- *Not obeying the single responsibility principle*—Our code is now responsible for both *creating* an `EmailSender` object and *using* it to send an email.
- *Considerable ceremony*—Of the 11 lines of code in the `RegisterUser` method, only the last two are doing anything useful. This makes it harder to read and harder to understand the intent of the method.
- *Tied to the implementation*—If you decide to refactor `EmailSender` and add another dependency, you'd need to update every place it's used. Likewise, if any of the *dependencies* are refactored, you would need to update this code too.

`UserController` has an *implicit* dependency on the `EmailSender` class, as it manually creates the object itself as part of the `RegisterUser` method. The only way to know that `UserController` uses `EmailSender` is to look at its source code. In contrast, `EmailSender` has *explicit* dependencies on `NetworkClient` and `MessageFactory`, which must be provided in the constructor. Similarly, `NetworkClient` has an *explicit* dependency on the `EmailServerSettings` class.

> **TIP** **Generally speaking, any dependencies in your code should be explicit, not implicit. Implicit dependencies are hard to reason about and difficult to test, so you should avoid them wherever you can. DI is useful for guiding you along this path.**

Dependency injection aims to solve the problem of building a dependency graph by *inverting* the chain of dependencies. Instead of the `UserController` creating its dependencies manually, deep inside the implementation details of the code, an already-created instance of `EmailSender` is injected via the constructor.

Now, obviously *something* needs to create the object, so the code to do that has to live somewhere. The service responsible for creating an object is called a *DI container* or an *IoC container*, as shown in figure 10.2.

> **DEFINITION** The ***DI container*** or ***IoC container*** is responsible for creating instances of services. It knows how to construct an instance of a service by creating all its dependencies and passing these to the constructor. I'll refer to it as a DI container throughout this book.

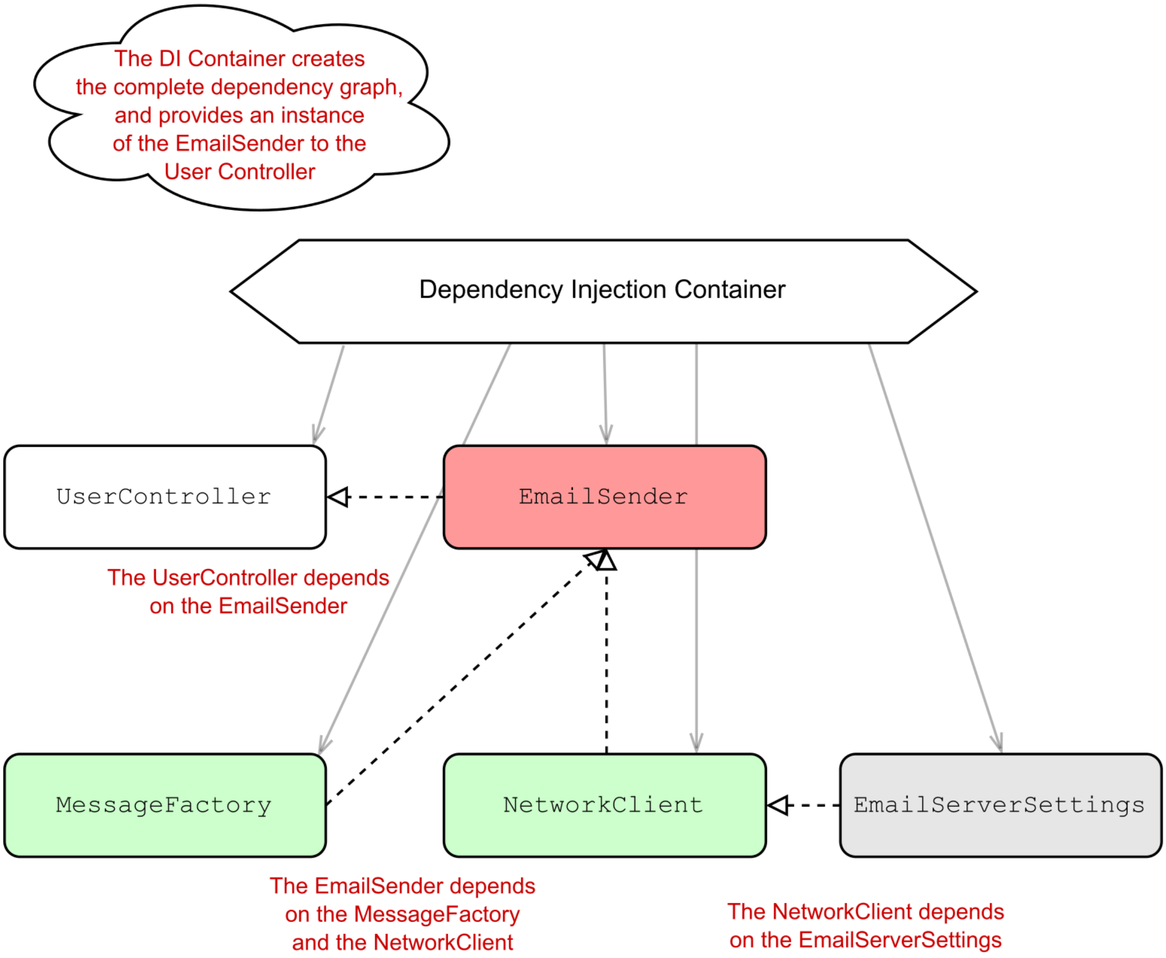

The DI Container creates the complete dependency graph, and provides an instance of the EmailSender to the User Controller
Dependency Injection Container
UserController
EmailSender
The UserController depends on the EmailSender
MessageFactory
NetworkClient
EmailServerSettings
The EmailSender depends on the MessageFactory and the NetworkClient
The NetworkClient depends on the EmailServerSettings

Figure 10.2 Dependency diagram using dependency injection. `UserController` **indirectly depends on all the other classes but doesn't need to know how to create them.** `UserController` **declares that it requires** `EmailSender` **and the container provides it.**

The term dependency injection is often used interchangeably with *inversion of control* (IoC). DI is a specific version of the more general principle of IoC. IoC describes the pattern where the *framework* calls your code to handle a request, instead of you writing the code to parse the request from bytes on the network card yourself. DI takes this further, where you allow the framework to create your dependencies too: instead of your `UserController` controlling how to create an `EmailSender` instance, it's provided one by the framework instead.

> **NOTE** Many DI containers are available for .NET: Autofac, Lamar, Unity, Ninject, Simple Injector… The list goes on! In chapter 19, you'll see how to replace the default ASP.NET Core container with one of these alternatives.

The advantage of adopting this pattern becomes apparent when you see how much it simplifies using dependencies. The following listing shows how `UserController` would look if you used DI to create `EmailSender` instead of doing it manually. All of the `new` cruft has gone, and you can focus purely on what the controller is doing—calling `EmailSender` and returning an `OkResult`.

Listing 10.4 Sending an email using DI to inject dependencies

```
public class UserController : ControllerBase
{
    private readonly EmailSender _emailSender;              #A
    public UserController(EmailSender emailSender)          #A
    {                                                       #A
        _emailSender = emailSender;                         #A
    }                                                       #A

    [HttpPost("register")]
    public IActionResult RegisterUser(string username)      #B
    {                                                       #B
        _emailSender.SendEmail(username);                   #B
        return Ok();                                        #B
    }
}
```

#A Instead of creating the dependencies implicitly, they're injected via the constructor.
#B The action method is easy to read and understand again.

One of the advantages of a DI container is that it has a single responsibility: creating objects or services. You ask a container for an instance of a service and it takes care of figuring out how to create the dependency graph, based on how you configure it.

> **NOTE** It's common to refer to *services* when talking about DI containers, which is slightly unfortunate as it's one of the most overloaded terms in software engineering! In this context, a service refers to any class or interface that the DI container creates when required.

The beauty of this approach is that by using explicit dependencies, you never have to write the mess of code you saw in listing 10.3. The DI container can inspect your service's constructor and work out how to write much of the code itself. DI containers are always configurable, so if you *want* to describe how to manually create an instance of a service you can, but by default you shouldn't need to.

> **TIP** You can inject dependencies into a service in other ways; for example, by using property injection. But constructor injection is the most common and is the only one supported out of the box in ASP.NET Core, so I'll only be using that in this book.

Hopefully, the advantages of using DI in your code are apparent from this quick example, but DI provides additional benefits that you get for free. In particular, it helps keep your code loosely coupled by coding to interfaces.

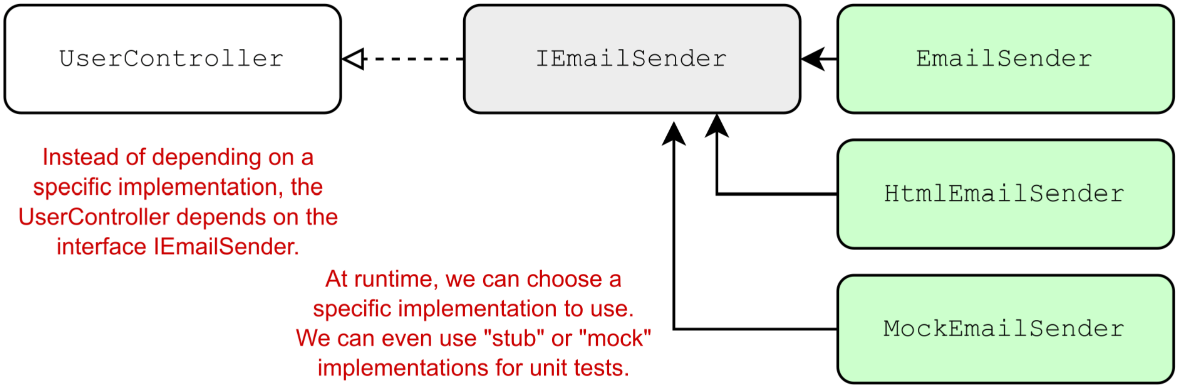
10.1.2 Creating loosely coupled code

Coupling is an important concept in object-oriented programming. It refers to how a given class depends on other classes to perform its function. Loosely coupled code doesn't need to know a lot of details about a particular component to use it.

The initial example of `UserController` and `EmailSender` was an example of tight coupling; you were creating the `EmailSender` object directly and needed to know exactly how to wire it up. On top of that, the code was difficult to test. Any attempts to test `UserController` would result in an email being sent. If you were testing the controller with a suite of unit tests, that seems like a surefire way to get your email server blacklisted for spam!

Taking `EmailSender` as a constructor parameter and removing the responsibility of creating the object helps reduce the coupling in the system. If the `EmailSender` implementation changes so that it has another dependency, you no longer have to update `UserController` at the same time.

One issue that remains is that `UserController` is still tied to an *implementation* rather than an *interface*. Coding to interfaces is a common design pattern that helps further reduce the coupling of a system, as you're not tied to a single implementation. This is particularly useful in making classes testable, as you can create "stub" or "mock" implementations of your dependencies for testing purposes, as shown in figure 10.3.



Instead of depending on a specific implementation, the UserController depends on the interface IEmailSender.

At runtime, we can choose a specific implementation to use. We can even use "stub" or "mock" implementations for unit tests.

Figure 10.3 By coding to interfaces instead of an explicit implementation, you can use different `IEmailSender` implementations in different scenarios, for example a `MockEmailSender` in unit tests.

TIP You can choose from many different mocking frameworks. My favorite is Moq, but NSubstitute and FakeItEasy are also popular options.

As an example, you might create an `IEmailSender` interface, which `EmailSender` would implement:

```
public interface IEmailSender
{
    public void SendEmail(string username);
}
```

`UserController` could then depend on this interface instead of the specific `EmailSender` implementation, as shown here. That would allow you to use a different implementation during unit tests, a `DummyEmailSender` for example.

Listing 10.5 Using interfaces with dependency injection

```
public class UserController : ControllerBase
{
    private readonly IEmailSender _emailSender;             #A
    public UserController(IEmailSender emailSender)         #A
    {                                                      #A
        _emailSender = emailSender;                        #A
    }                                                      #A

    [HttpPost("register")]
       public IActionResult RegisterUser(string username)
    {
        _emailSender.SendEmail(username);                  #B
        return Ok();
    }
}
```

#A You now depend on IEmailSender instead of the specific EmailSender implementation.
#B You don't care what the implementation is, as long as it implements IEmailSender.

The key point here is that the consuming code, `UserController`, doesn't care how the dependency is implemented, only that it implements the `IEmailSender` interface and exposes a `SendEmail` method. The application code is now independent of the implementation.

Hopefully, the principles behind DI seem sound—by having loosely coupled code, it's easy to change or swap out implementations completely. But this still leaves you with a question: how does the application know to use `EmailSender` in production instead of `DummyEmailSender`? The process of telling your DI container, "when you need `IEmailSender`, use `EmailSender`" is called *registration*.

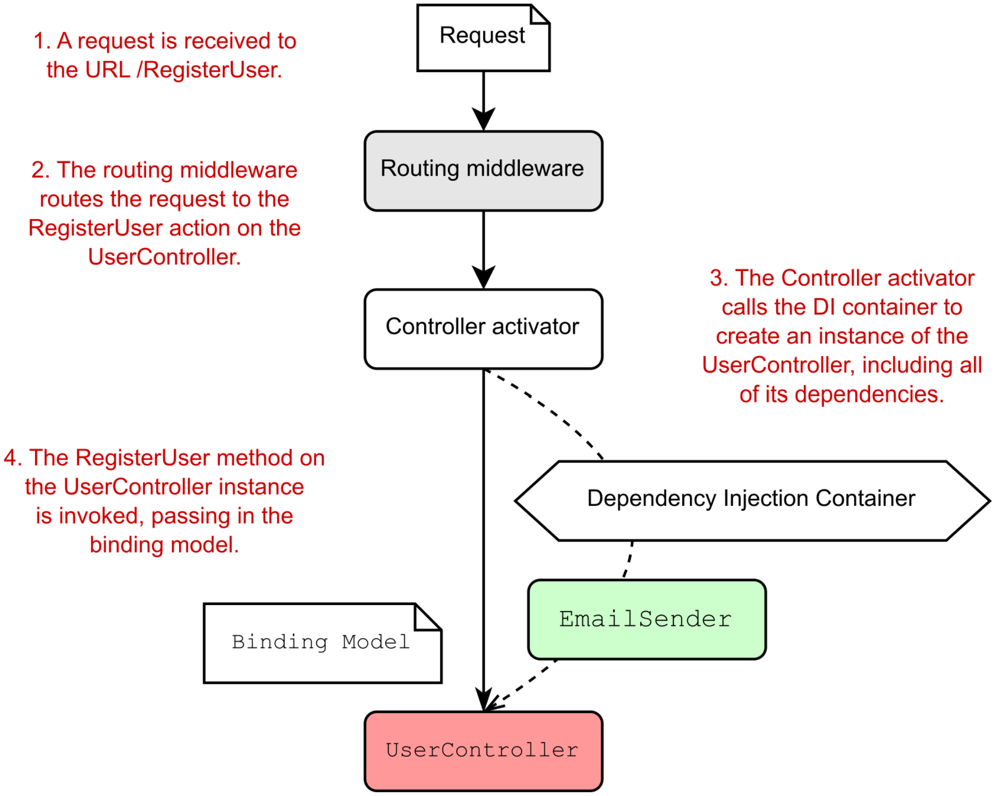
> **DEFINITION** **You *register* services with a DI container so that it knows which implementation to use for each requested service. This typically takes the form of, "for interface X, use implementation Y."**

Exactly how you register your interfaces and types with a DI container can vary depending on the specific DI container implementation, but the principles are generally all the same. ASP.NET Core includes a simple DI container out of the box, so let's look at how it's used during a typical request.

10.1.3 Dependency injection in ASP.NET Core

ASP.NET Core was designed from the outset to be modular and composable, with an almost plugin-style architecture, which is generally complemented by DI. Consequently, ASP.NET Core includes a simple DI container that all the framework libraries use to register themselves and their dependencies.

This container is used, for example, to register the Razor Pages and Web API infrastructure—the formatters, the view engine, the validation system, and so on. It's only a basic container, so it only exposes a few methods for registering services, but you can also replace it with a third-party DI container. This can give you extra capabilities, such as auto-registration or setter injection. The DI container is built into the ASP.NET Core hosting model, as shown in figure 10.4.



1. A request is received to the URL /RegisterUser.
Request
2. The routing middleware routes the request to the RegisterUser action on the UserController.
Routing middleware
Controller activator
3. The Controller activator calls the DI container to create an instance of the UserController, including all of its dependencies.
4. The RegisterUser method on the UserController instance is invoked, passing in the binding model.
Dependency Injection Container
EmailSender
Binding Model
UserController

Figure 10.4 The ASP.NET Core hosting model uses the DI container to fulfill dependencies when creating controllers.

The hosting model pulls dependencies from the DI container when they're needed. If the framework determines that UserController is required due to the incoming URL/route, the controller activator responsible for creating an API controller instance will ask the DI container for an IEmailSender implementation.

NOTE This approach, where a class calls the DI container directly to ask for a class is called the *service locator* pattern. Generally speaking, you should try to avoid this pattern in your code; include your dependencies as constructor arguments directly and let the DI container provide them for you.[45]

The DI container needs to know what to create when asked for IEmailSender, so you must have registered an implementation, such as EmailSender, with the container. Once an implementation is registered, the DI container can inject it anywhere. That means you can inject framework-related services into your own custom services, as long as they are registered with the container. It also means you can register alternative versions of framework services and have the framework automatically use those in place of the defaults.

The flexibility to choose exactly how and which components you combine in your applications is one of the selling points of DI. In the next section, you'll learn how to configure DI in your own ASP.NET Core application, using the default, built-in container.

10.2 Using the dependency injection container

In previous versions of ASP.NET, using dependency injection was entirely optional. In contrast, to build all but the most trivial ASP.NET Core apps, some degree of DI is required. As I've mentioned, the underlying framework depends on it, so things like using Razor Pages and API controllers require you to configure the required services.

In this section, you'll see how to register these framework services with the built-in container, as well as how to register your own services. Once services are registered, you can use them as dependencies and inject them into any of the services in your application.

10.2.1 Adding ASP.NET Core framework services to the container

As I described earlier, ASP.NET Core uses DI to configure its internal components as well as your own custom services. To use these components at runtime, the DI container needs to know about all the classes it will need. You register these in the ConfigureServices method of your Startup class.

NOTE The dependency injection container is set up in the ConfigureServices method of your Startup class in Startup.cs.

[45] You can read about the Service Locator antipattern in *Dependency Injection Principles, Practices, and Patterns* by Steven van Deursen and Mark Seemann (Manning, 2019) https://livebook.manning.com/book/dependency-injection-principles-practices-patterns/chapter-5/section-5-2.

Now, if you're thinking, "Wait, I have to configure the internal components myself?" then don't panic. Although true in one sense—you do need to explicitly register the components with the container in your app—all the libraries you'll use expose handy extension methods to take care of the nitty-gritty details for you. These extension methods configure everything you'll need in one fell swoop, instead of leaving you to manually wire everything up.

For example, the Razor Pages framework exposes the `AddRazorPages()` extension method that you saw in chapters 2, 3, and 4. Invoke the extension method in `ConfigureServices` of `Startup`.

Listing 10.6 Registering the MVC services with the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();           #A
}
```

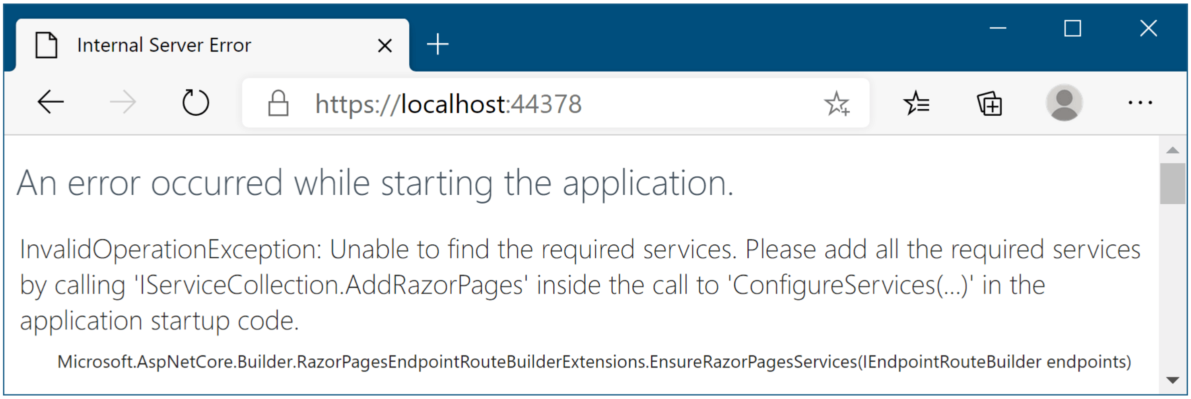
#A The AddRazorPages extension method adds all necessary services to the IServiceCollection.

It's as simple as that. Under the hood, this call is registering multiple components with the DI container, using the same APIs you'll see shortly for registering your own services.

> **TIP The `AddControllers()` method registers the required services for API controllers, as you saw in chapter 9. There is a similar method, `AddControllersWithViews()` if you're using MVC controllers with Razor views, and an `AddMvc()` method to add all of them, and the kitchen sink!**

Most nontrivial libraries that you add to your application will have services that you need to add to the DI container. By convention, each library that has necessary services should expose an `Add*()` extension method that you can call in `ConfigureServices`.

There's no way of knowing exactly which libraries will require you to add services to the container, it's generally a case of checking the documentation for any libraries you use. If you forget to add them, then you may find the functionality doesn't work, or you might get a handy exception like the one shown in figure 10.5. Keep an eye out for these and be sure to register any services that you need!



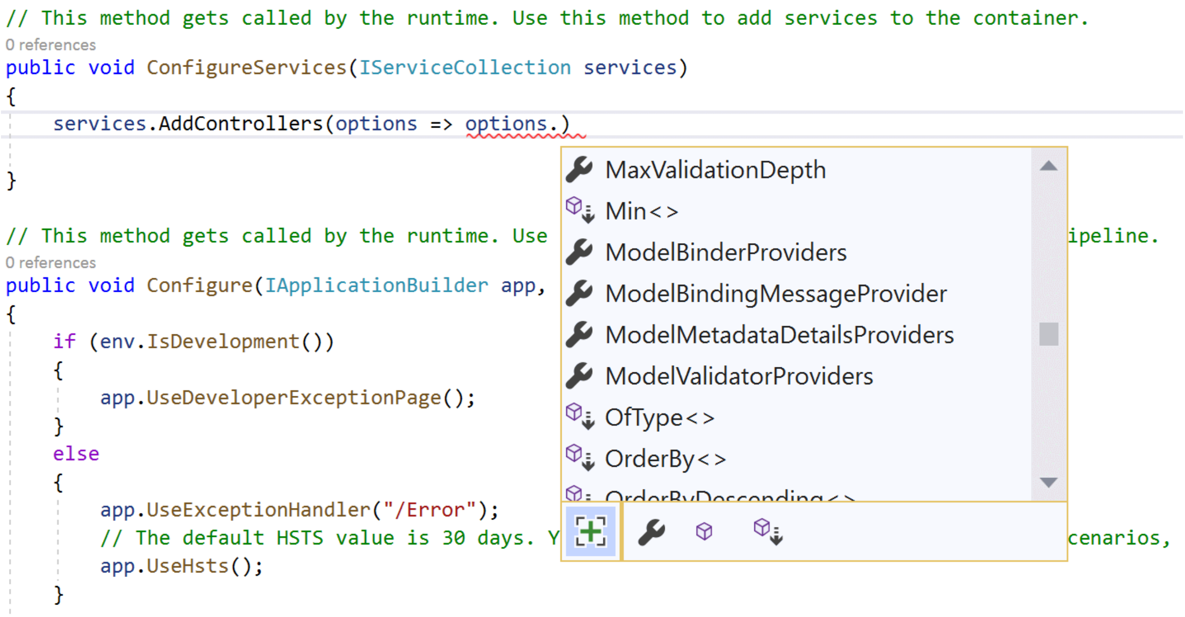
An error occurred while starting the application.

InvalidOperationException: Unable to find the required services. Please add all the required services by calling 'IServiceCollection.AddRazorPages' inside the call to 'ConfigureServices(...)' in the application startup code.

Microsoft.AspNetCore.Builder.RazorPagesEndpointRouteBuilderExtensions.EnsureRazorPagesServices(IEndpointRouteBuilder endpoints)

Figure 10.5 If you fail to call `AddRazorPages` in the `ConfigureServices` of `Startup`, you'll get a friendly exception message at runtime.

It's also worth noting that some of the `Add*()` extension methods allow you to specify additional options when you call them, often by way of a lambda expression. You can think of these as configuring the installation of a service into your application. The `AddControllers` method, for example, provides a wealth of options for fine-tuning its behavior if you want to get your fingers dirty, as shown by the IntelliSense snippet in figure 10.6.



```
// This method gets called by the runtime. Use this method to add services to the container.
0 references
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(options => options.)

}

// This method gets called by the runtime. Use                                           ipeline.
0 references
public void Configure(IApplicationBuilder app,
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        // The default HSTS value is 30 days. Y                                          cenarios,
        app.UseHsts();
    }
```

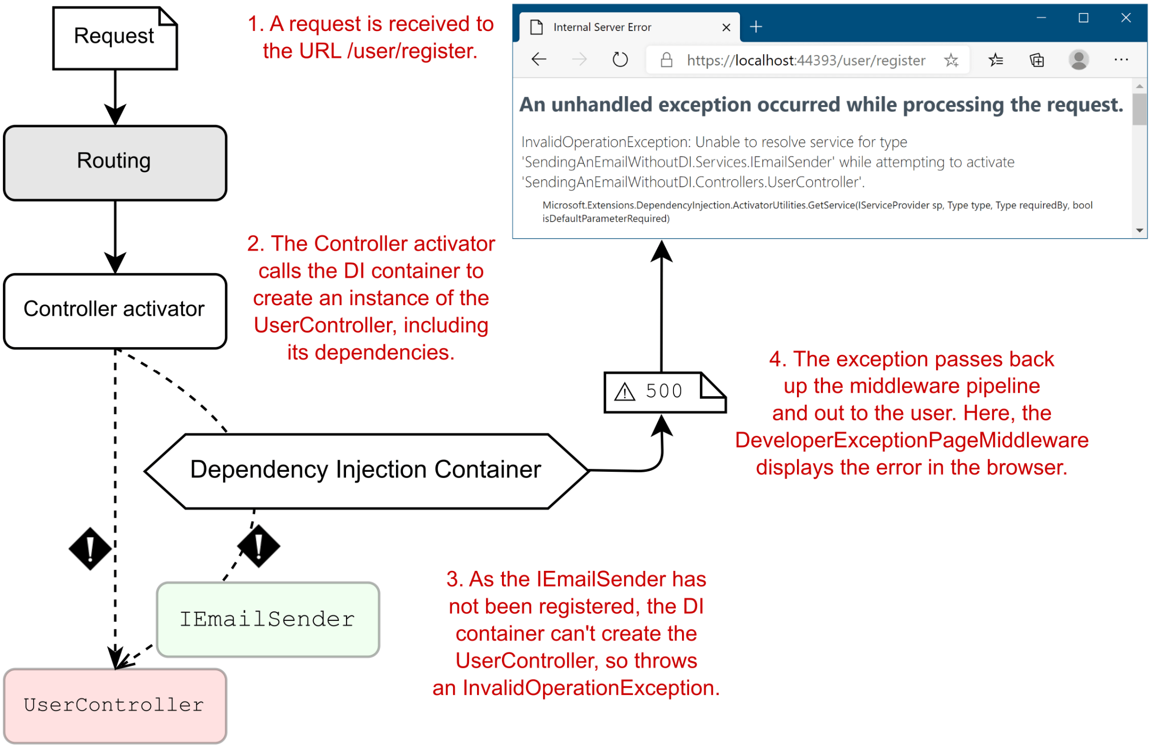
Figure 10.6 Configuring services when adding them to the service collection. The `AddControllers()` function allows you to configure a wealth of the internals of the API controller services. Similar configuration options are available in the `AddRazorPages()` function.

Once you've added the required framework services, you can get down to business and register your own services, so you can use DI in your own code.

10.2.2 Registering your own services with the container

In the first section of this chapter, I described a system for sending emails when a new user registers on your application. Initially, `UserController` was manually creating an instance of `EmailSender`, but you subsequently refactored this, so you inject an instance of `IEmailSender` into the constructor instead.

The final step to make this refactoring work is to configure your services with the DI container. This lets the DI container know what to use when it needs to fulfill the `IEmailSender` dependency. If you don't register your services, then you'll get an exception at runtime, like the one in figure 10.7. Luckily, this exception is useful, letting you know which service wasn't registered (`IEmailSender`) and which service needed it (`UserController`).



1. A request is received to the URL /user/register.

2. The Controller activator calls the DI container to create an instance of the UserController, including its dependencies.

3. As the IEmailSender has not been registered, the DI container can't create the UserController, so throws an InvalidOperationException.

4. The exception passes back up the middleware pipeline and out to the user. Here, the DeveloperExceptionPageMiddleware displays the error in the browser.

An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type 'SendingAnEmailWithoutDI.Services.IEmailSender' while attempting to activate 'SendingAnEmailWithoutDI.Controllers.UserController'.

Microsoft.Extensions.DependencyInjection.ActivatorUtilities.GetService(IServiceProvider sp, Type type, Type requiredBy, bool isDefaultParameterRequired)

Figure 10.7 If you don't register all your required dependencies in `ConfigureServices`, you'll get an exception at runtime, telling you which service wasn't registered.

In order to completely configure the application, you need to register `EmailSender` and all of its dependencies with the DI container, as shown in figure 10.8.

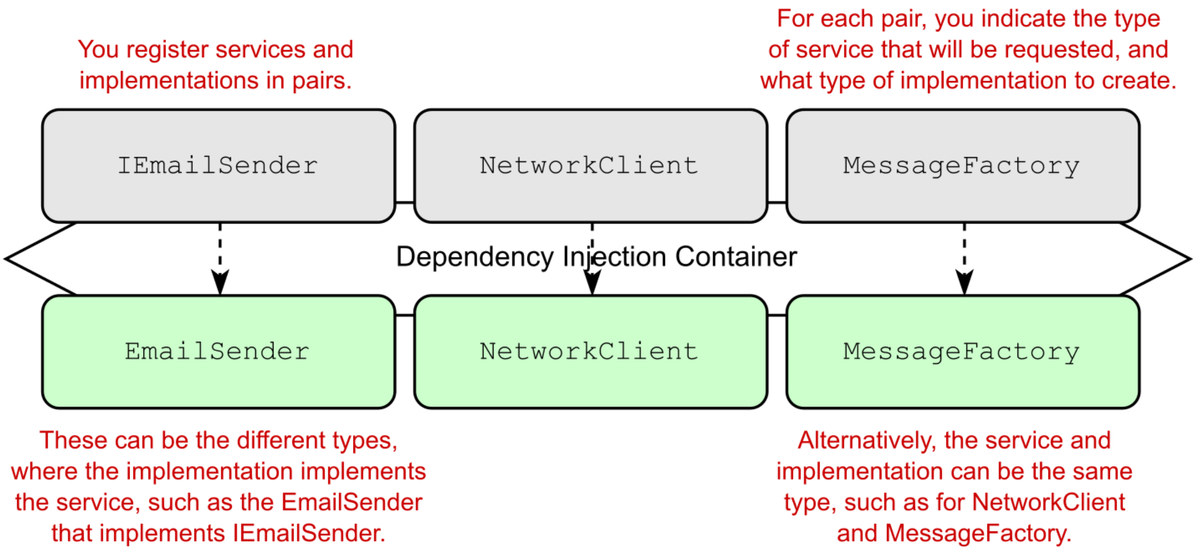

You register services and implementations in pairs.
For each pair, you indicate the type of service that will be requested, and what type of implementation to create.
IEmailSender
NetworkClient
MessageFactory
Dependency Injection Container
EmailSender
NetworkClient
MessageFactory
These can be the different types, where the implementation implements the service, such as the EmailSender that implements IEmailSender.
Alternatively, the service and implementation can be the same type, such as for NetworkClient and MessageFactory.

Figure 10.8 Configuring the DI container in your application involves telling it what type to use when a given service is requested; for example, "Use `EmailSender` when `IEmailSender` is required."

Configuring DI consists of making a series of statements about the services in your app. For example:

- When a service requires `IEmailSender`, use an instance of `EmailSender`
- When a service requires `NetworkClient`, use an instance of `NetworkClient`
- When a service requires `MessageFactory`, use an instance of `MessageFactory`

> **NOTE** **You'll also need to register the `EmailServerSettings` object with the DI container—we'll do that slightly differently, in the next section.**

These statements are made by calling various `Add*` methods on `IServiceCollection` in the `ConfigureServices` method. Each method provides three pieces of information to the DI container:

- *Service type*—`TService`. This is the class or interface that will be requested as a dependency. Often an interface, such as `IEmailSender`, but sometimes a concrete type, such as `NetworkClient` or `MessageFactory`.
- *Implementation type*—`TService` or `TImplementation`. This is the class the container should create to fulfill the dependency. Must be a concrete type, such as `EmailSender`. May be the same as the service type, as for `NetworkClient` and `MessageFactory`.
- *Lifetime—Transient, Singleton, or Scoped*. This defines how long an instance of the service should be used for. I'll discuss lifetimes in detail in section 10.3.

The following listing shows how you can configure EmailSender and its dependencies in your application using three different methods: AddScoped<TService>, AddSingleton<TService>, and AddScoped<TService, TImplementation>. This tells the DI container how to create each of the TService instances when they're required.

Listing 10.7 Registering services with the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();                              #A

    services.AddScoped<IEmailSender, EmailSender>();        #B
    services.AddScoped<NetworkClient>();                    #C
    services.AddSingleton<MessageFactory>();                #D
}
```

#A You're using API controllers, so must call AddControllers.
#B Whenever you require an IEmailSender, use EmailSender.
#C Whenever you require a NetworkClient, use NetworkClient.
#D Whenever you require a MessageFactory, use MessageFactory.

And that's all there is to dependency injection! It may seem a little bit like magic,[46] but you're just giving the container instructions on how to make all the constituent parts. You give it a recipe for how to cook the chili, shred the lettuce, and grate the cheese, so that when you ask for a burrito, it can put all the parts together and hand you your meal!

The service type and implementation type are the same for NetworkClient and MessageFactory, so there's no need to specify the same type twice in the AddScoped method, hence the slightly simpler signature. These generic methods aren't the only way to register services with the container, you can also provide objects directly, or by using lambdas, as you'll see in the next section.

10.2.3 Registering services using objects and lambdas

As I mentioned earlier, I didn't *quite* register all the services required by UserController. In all my previous examples, NetworkClient depends on EmailServerSettings, which you'll also need to register with the DI container for your project to run without exceptions.

I avoided registering this object in the preceding example because you have to use a slightly different approach. The preceding Add* methods use generics to specify the Type of the class to register, but they don't give any indication of *how* to construct an instance of that type. Instead, the container makes a number of assumptions that you have to adhere to:

- The class must be a concrete type.

[46] Under the hood, the built-in ASP.NET Core DI container uses optimized reflection to create dependencies, but different DI containers may use other approaches.

- The class must only have a single "valid" constructor that the container can use.
- For a constructor to be "valid," all constructor arguments must be registered with the container, or must be an argument with a default value.

> **NOTE** **These limitations apply to the simple built-in DI container. If you choose to use a third-party container in your app, then they may have a different set of limitations.**

The `EmailServerSettings` class doesn't meet these requirements, as it requires you to provide a `host` and `port` in the constructor, which are `string`s without default values:

```
public class EmailServerSettings
{
    public EmailServerSettings(string host, int port)
    {
        Host = host;
        Port = port;
    }
    public string Host { get; }
    public int Port { get; }
}
```

You can't register these primitive types in the container; it would be weird to say, "For every `string` constructor argument, in any type, use the `"smtp.server.com"` value!"

Instead, you can create an instance of the `EmailServerSettings` object yourself and provide *that* to the container, as shown next. The container uses the pre-constructed object whenever an instance of the `EmailServerSettings` object is required.

Listing 10.8 Providing an object instance when registering services

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddScoped<IEmailSender, EmailSender>();
    services.AddSingleton<NetworkClient>();
    services.AddScoped<MessageFactory>();
    services.AddSingleton(
        new EmailServerSettings          #A
        (                                #A
            host: "smtp.server.com",     #A
            port: 25                     #A
        ));                              #A
}
```

#A This instance of EmailServerSettings will be used whenever an instance is required.

This works fine if you only want to have a single instance of `EmailServerSettings` in your application—*the same object* will be shared everywhere. But what if you want to create a *new* object each time one is requested?

NOTE When the same object is used whenever it's requested, it's known as a singleton. I'll discuss singletons along with other *lifetimes* in detail in the next section. The lifetime is how long the DI container should use a given object to fulfill a service's dependencies.

Instead of providing a single instance that the container will always use, you can also provide a *function* that the container invokes when it needs an instance of the type, as shown in figure 10.9.

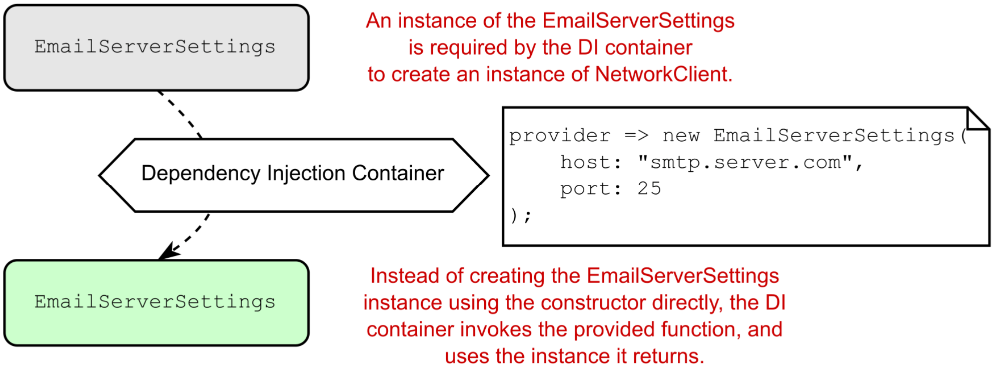

EmailServerSettings
An instance of the EmailServerSettings is required by the DI container to create an instance of NetworkClient.
Dependency Injection Container
provider => new EmailServerSettings(
    host: "smtp.server.com",
    port: 25
);
EmailServerSettings
Instead of creating the EmailServerSettings instance using the constructor directly, the DI container invokes the provided function, and uses the instance it returns.

Figure 10.9 You can register a function with the DI container that will be invoked whenever a new instance of a service is required.

The easiest way to do this is to use a lambda function (an anonymous delegate), in which the container creates a new `EmailServerSettings` object when it's needed.

Listing 10.9 Using a lambda factory function to register a dependency

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddScoped<IEmailSender, EmailSender>();
    services.AddSingleton<NetworkClient>();
    services.AddScoped<MessageFactory>();
    services.AddScoped(                           #A
        provider =>                                  #B
            new EmailServerSettings             #C
            (                                   #C
                host: "smtp.server.com",        #C
                port: 25                        #C
            ));                                 #C
}
```

#A Because you're providing a function to create the object, you aren't restricted to singleton.
#B The lambda is provided an instance of IServiceProvider.
#C The constructor is called every time an EmailServerSettings is required, instead of only once.

In this example, I've changed the lifetime of the created `EmailServerSettings` object to be *scoped* instead of *singleton* and provided a factory lambda function that returns a new `EmailServerSettings` object. Every time the container requires a new `EmailServerSettings`, it executes the function and uses the new object it returns.

> **NOTE** I'll discuss lifetimes shortly, but it's important to notice that there are two concepts of a singleton here. If you create an object and pass it to the container, it's *always* registered as a singleton. You can also register *any* arbitrary class as a singleton and the container will only use one instance throughout your application.

When you use a lambda to register your services, you're provided with an `IServiceProvider` instance at runtime, called `provider` in listing 10.9. This is the public API of the DI container itself which exposes the `GetService()` function. If you need to obtain dependencies to create an instance of your service, you can reach into the container at runtime in this way, but you should avoid doing so if possible.

> **TIP** Avoid calling `GetService()` in your factory functions if possible. Instead, favor constructor injection—it's more performant, as well as being simpler to reason about.

Open generics and dependency injection

As already mentioned, you couldn't use the generic registration methods with `EmailServerSettings` because it uses primitive dependencies (in this case, `string`) in its constructor. You also can't use the generic registration methods to register *open generics*.

Open generics are types that contain a generic type parameter, such as `Repository <T>`. You normally use this sort of type to define a base behavior that you can use with multiple generic types. In the `Repository<T>` example, you might inject `IRepository<Customer>` into your services, which should inject an instance of `DbRepository<Customer>`, for example.

To register these types, you must use a different overload of the `Add*` methods. For example,

```
services.AddScoped(typeof(IRespository<>), typeof(DbRepository<>));
```

This ensures that whenever a service constructor requires `IRespository<T>`, the container injects an instance of `DbRepository<T>`.

At this point, all your dependencies are registered. But `ConfigureServices` is starting to look a little messy, isn't it? It's entirely down to personal preference, but I like to group my services into logical collections and create extension methods for them, as in the following listing. This creates an equivalent of the framework's `AddControllers()` extension method—a nice, simple, registration API. As you add more and more features to your app, I think you'll appreciate it too.

Listing 10.10 Creating an extension method to tidy up adding multiple services

```
public static class EmailSenderServiceCollectionExtensions
{
    public static IServiceCollection AddEmailSender(
        this IServiceCollection services)                   #A
    {
        services.AddScoped<IEmailSender, EmailSender>();     #B
        services.AddSingleton<NetworkClient>();              #B
        services.AddScoped<MessageFactory>();                #B
        services.AddSingleton(                               #B
            new EmailServerSettings                          #B
            (                                                #B
                host: "smtp.server.com",                     #B
                port: 25                                     #B
            ));                                              #B
        return services;                                     #C
    }
}
```

#A Extend the IServiceCollection that's provided in the ConfigureServices method.
#B Cut and paste your registration code from ConfigureServices.
#C By convention, return the IServiceCollection to allow method chaining.

With the preceding extension method created, the `ConfigureServices` method is much easier to grok!

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddEmailSender();
}
```

So far, you've seen how to register the simple DI cases where you have a single implementation of a service. In some scenarios, you might find you have multiple implementations of an interface. In the next section, you'll see how to register these with the container to match your requirements.

10.2.4 Registering a service in the container multiple times

One of the advantages of coding to interfaces is that you can create multiple implementations of a service. For example, imagine you want to create a more generalized version of `IEmailSender` so that you can send messages via SMS or Facebook, as well as by email. You create an interface for it

```
public interface IMessageSender
{
    public void SendMessage(string message);
}
```

as well as several implementations: `EmailSender`, `SmsSender`, and `FacebookSender`. But how do you register these implementations in the container? And how can you inject these implementations into your UserController? The answer varies slightly depending if you want to use all the implementations in your consumer or only one of them.

INJECTING MULTIPLE IMPLEMENTATIONS OF AN INTERFACE

Imagine you want to send a message using each of the IMessageSender implementations whenever a new user registers, so they get an email, an SMS, and a Facebook message, as shown in figure 10.10.

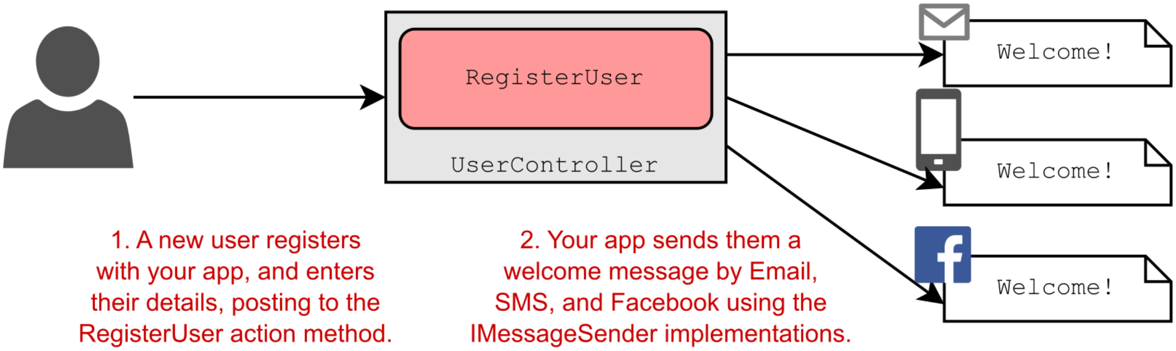
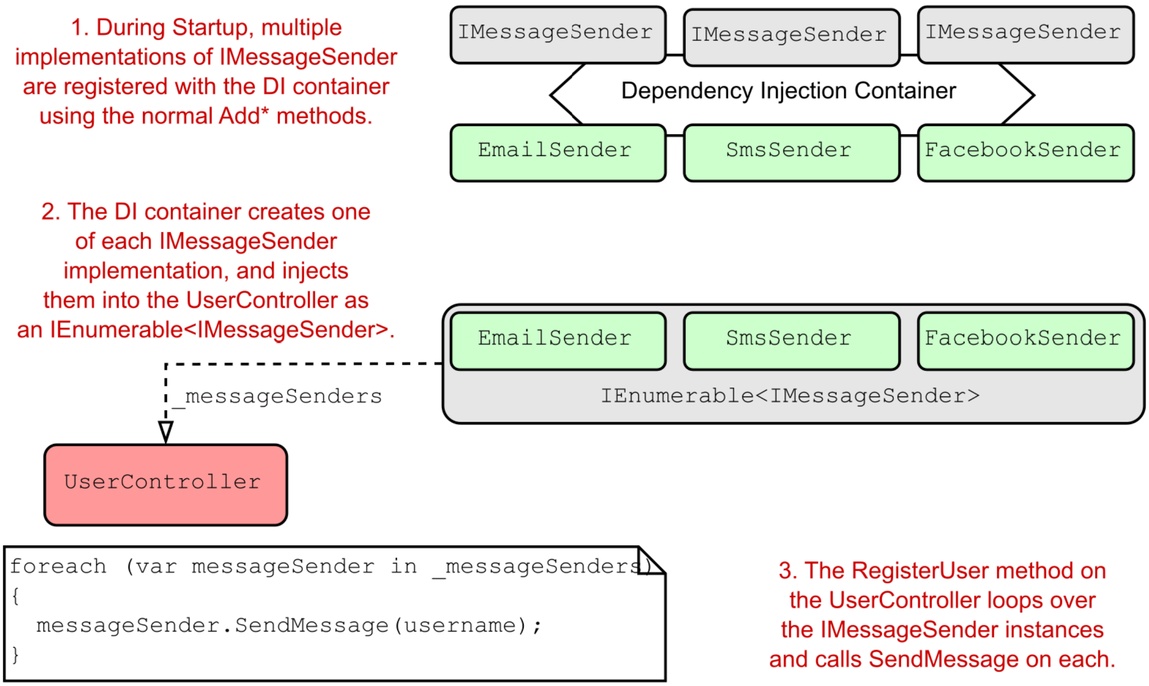

RegisterUser
UserController
Welcome!
Welcome!
Welcome!
1. A new user registers with your app, and enters their details, posting to the RegisterUser action method.
2. Your app sends them a welcome message by Email, SMS, and Facebook using the IMessageSender implementations.

Figure 10.10 When a user registers with your application, they call the RegisterUser method. This sends them an email, an SMS, and a Facebook message using the IMessageSender classes.

The easiest way to achieve this is to register all the service implementations in your DI container and have it inject one of each type into UserController. UserController can then use a simple foreach loop to call SendMessage() on each implementation, as in figure 10.11.


1. During Startup, multiple implementations of IMessageSender are registered with the DI container using the normal Add* methods.
IMessageSender
IMessageSender
IMessageSender
Dependency Injection Container
EmailSender
SmsSender
FacebookSender
2. The DI container creates one of each IMessageSender implementation, and injects them into the UserController as an IEnumerable<IMessageSender>.
EmailSender
SmsSender
FacebookSender
IEnumerable<IMessageSender>
_messageSenders
UserController
3. The RegisterUser method on the UserController loops over the IMessageSender instances and calls SendMessage on each.

```
foreach (var messageSender in _messageSenders)
{
  messageSender.SendMessage(username);
}
```

Figure 10.11 You can register multiple implementations of a service with the DI container, such as `IEmailSender` in this example. You can retrieve an instance of each of these implementations by requiring `IEnumerable<IMessageSender>` in the `UserController` constructor.

You register multiple implementations of the same service with a DI container in exactly the same way as for single implementations, using the `Add*` extension methods. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddScoped<IMessageSender, EmailSender>();
    services.AddScoped<IMessageSender, SmsSender>();
    services.AddScoped<IMessageSender, FacebookSender>();
}
```

You can then inject `IEnumerable<IMessageSender>` into `UserController`, as shown in the following listing. The container injects an array of `IMessageSender` containing one of each of the implementations you have registered, in the same order as you registered them. You can then use a standard `foreach` loop in the `RegisterUser` method to call `SendMessage` on each implementation.

Listing 10.11 Injecting multiple implementations of a service into a consumer

```
public class UserController : ControllerBase
```

```
{
    private readonly IEnumerable<IMessageSender> _messageSenders;     #A
    public UserController(                                            #A
        IEnumerable<IMessageSender> messageSenders)                   #A
    {                                                                 #A
        _messageSenders = messageSenders;                             #A
    }                                                                 #A

    [HttpPost("register")]
    public IActionResult RegisterUser(string username)
    {
        foreach (var messageSender in _messageSenders)             #B
        {                                                         #B
            messageSender.SendMessage(username);                  #B
        }                                                         #B

        return Ok();
    }
}
```

#A Requesting an IEnumerable will inject an array of IMessageSender.
#B Each IMessageSender in the IEnumerable is a different implementation.

> **WARNING** You must use `IEnumerable<T>` as the constructor argument to inject all the registered types of a service, `T`. Even though this will be injected as a `T[]` array, you can't use `T[]` or `ICollection<T>` as your constructor argument. Doing so will cause an `InvalidOperationException`, similar to that in figure 10.7.

It's simple enough to inject all the registered implementations of a service, but what if you only need one? How does the container know which one to use?

INJECTING A SINGLE IMPLEMENTATION WHEN MULTIPLE SERVICES ARE REGISTERED

Imagine you've already registered all the `IMessageSender` implementations, what happens if you have a service that requires only one of them? For example

```
public class SingleMessageSender
{
    private readonly IMessageSender _messageSender;
    public SingleMessageSender(IMessageSender messageSender)
    {
        _messageSender = messageSender;
    }
}
```

The container needs to pick a *single* `IMessageSender` to inject into this service, out of the three implementations available. It does this by using the *last* registered implementation—the `FacebookSender` from the previous example.

> **NOTE** The DI container will use the last registered implementation of a service when resolving a single instance of the service.

This can be particularly useful for replacing built-in DI registrations with your own services. If you have a custom implementation of a service that you know is registered within a library's `Add*` extension method, you can override that registration by registering your own implementation afterwards. The DI container will use your implementation whenever a single instance of the service is requested.

The main disadvantage with this approach is that you still end up with *multiple* implementations registered—you can inject an `IEnumerable<T>` as before. Sometimes you want to conditionally register a service, so you only ever have a single registered implementation.

CONDITIONALLY REGISTERING SERVICES USING TRYADD

Sometimes, you'll only want to add an implementation of a service if one hasn't already been added. This is particularly useful for library authors; they can create a default implementation of an interface and only register it if the user hasn't already registered their own implementation.

You can find several extension methods for conditional registration in the `Microsoft.Extensions.DependencyInjection.Extensions` namespace, such as `TryAddScoped`. This checks to make sure a service hasn't been registered with the container before calling `AddScoped` on the implementation. The following listing shows you conditionally adding `SmsSender`, only if there are no existing `IMessageSender` implementations. As you previously registered `EmailSender`, the container will ignore the `SmsSender` registration, so it won't be available in your app.

Listing 10.12 Conditionally adding a service using `TryAddScoped`

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IMessageSender, EmailSender>();    #A
    services.TryAddScoped<IMessageSender, SmsSender>();   #B
}
```

#A EmailSender is registered with the container.
#B There's already an IMessageSender implementation, so SmsSender isn't registered.

Code like this often doesn't make a lot of sense at the application level, but it can be useful if you're building libraries for use in multiple apps. The ASP.NET Core framework, for example, uses `TryAdd*` in many places, which lets you easily register alternative implementations of internal components in your own application if you want.

You can also *replace* a previously registered implementation by using the `Replace()` extension method. Unfortunately, the API for this method isn't as friendly as the `TryAdd` methods. To replace a previously registered `IMessageSender` with the `SmsSender`, you would use

```
services.Replace(new ServiceDescriptor(
    typeof(IMessageSender), typeof(SmsSender), ServiceLifetime.Scoped
));
```

> **TIP** When using Replace, you must provide the same lifetime as was used to register the service that is being replaced.

That pretty much covers registering dependencies. Before we look in more depth at the "lifetime" aspect of dependencies, we'll take a quick detour and look at two ways other than a constructor to inject dependencies in your app.

10.2.5 Injecting services into action methods, page handlers, and views

I mentioned in section 10.1 that the ASP.NET Core DI container only supports constructor injection, but there are three additional locations where you can use dependency injection:

1. Action methods
2. Page handler methods
3. View templates

In this section, I'll briefly discuss these three situations, how they work, and when you might want to use them.

INJECTING SERVICES DIRECTLY INTO ACTION METHODS AND PAGE HANDLERS USING [FROMSERVICES]

API controllers typically contain multiple action methods that logically belong together. You might group all the action methods related to managing user accounts into the same controller, for example. This allows you to apply filters and authorization to all the action methods collectively, as you'll see in chapter 13.

As you add additional action methods to a controller, you may find the controller needs additional services to implement new action methods. With constructor injection, all these dependencies are provided via the constructor. That means the DI container must create *all* the dependencies for *every* action method in a controller, even if none of them are required by the action method being called.

Consider listing 10.13 for example. This shows `UserController` with two stub methods: `RegisterUser` and `PromoteUser`. Each action method requires a different dependency, so both dependencies will be created and injected, whichever action method is called by the request. If `IPromotionService` or `IMessageSender` have lots of dependencies themselves, the DI container may have to create lots of objects for a service that is often not used.

Listing 10.13 Injecting services into a controller via the constructor

```
public class UserController : ControllerBase
{
    private readonly IMessageSender _messageSender;          #A
    private readonly IPromotionService _promoService;         #A
    public UserController(
        IMessageSender messageSender, IPromotionService promoService)
    {
        _messageSender = messageSender;
        _promoService = promoService;
    }

    [HttpPost("register")]
    public IActionResult RegisterUser(string username)
    {
        _messageSender.SendMessage(username);         #B
        return Ok();
    }

    [HttpPost("promote")]
    public IActionResult PromoteUser(string username, int level)
    {
        _promoService.PromoteUser(username, level);      #C
        return Ok();
    }
}
```

#A Both IMessageSender and IPromotionService are injected into the constructor every time.
#B The RegisterUser method only uses IMessageSender.
#C The PromoteUser method only uses IPromotionService.

If you know a service is particularly expensive to create, you can choose to inject it as a dependency *directly* into the action method, instead of into the controller's constructor. This ensures the DI container only creates the dependency when the *specific* action method is invoked, as opposed to when *any* action method on the controller is invoked.

> **NOTE** Generally speaking, your controllers should be sufficiently cohesive that this approach isn't necessary. If you find you have a controller that's dependent on many services, each of which is used by a single action method, you might want to consider splitting up your controller.

You can directly inject a dependency into an action method by passing it as a parameter to the method and using the `[FromServices]` attribute. During model binding, the framework will resolve the parameter from the DI container, instead of from the request values. This listing shows how you could rewrite listing 10.13 to use `[FromServices]` instead of constructor injection.

Listing 10.14 Injecting services into a controller using the `[FromServices]` attribute

```
public class UserController : ControllerBase
{
    [HttpPost("register")]
    public IActionResult RegisterUser(                    #A
        [FromServices] IMessageSender messageSender,      #A
        string username)                                  #A
    {
        messageSender.SendMessage(username);              #B
        return Ok();
    }
    [HttpPost("promote")]
    public IActionResult PromoteUser(                     #C
        [FromServices] IPromotionService promoService,    #C
        string username, int level)                       #C
    {
        promoService.PromoteUser(username, level);        #D
        return Ok();
    }
}
```

#A The [FromServices] attribute ensures IMessageSender is resolved from the DI container.
#B IMessageSender is only available in RegisterUser.
#C IPromotionService is resolved from the DI container and injected as a parameter.
#D Only the PromoteUser method can use IPromotionService.

You might be tempted to use the `[FromServices]` attribute in all your action methods, but I'd encourage you to use standard constructor injection most of the time. Having the constructor as a single location that declares all the dependencies of a class can be useful, so I only use `[FromServices]` in the rare cases where creating an instance of a dependency is expensive and is only used in a single action method.

The `[FromServices]` attribute can be used in exactly the same way with Razor Pages. You can inject services into a Razor Page's page handler, instead of into the constructor, as shown in the following listing.

> **TIP** Just because you *can* inject services into page handlers like this, doesn't mean you *should*. Razor Pages are inherently designed to be small and cohesive, so it's better to just use constructor injection.

Listing 10.15 Injecting services into a Razor Page using the `[FromServices]` attribute

```
public class PromoteUserModel: PageModel
{
    public void OnGet()                                   #A
    {
    }

    public IActionResult OnPost(                          #B
        [FromServices] IPromotionService promoService,    #B
        string username, int level)                       #B
    {
        promoService.PromoteUser(username, level);        #C
        return RedirectToPage("success");
    }

}
```

#A The OnGet handler does not require any services
#C IPromotionService is resolved from the DI container and injected as a parameter.
#D Only the OnPost page handler can use IPromotionService.

Generally speaking, if you find you need to use the `[FromServices]` attribute then you should step back and look at your controller/Razor Page. It's likely that you're trying to do too much in one class. Instead of working around the issue with `[FromServices]`, consider splitting the class up, or pushing some behavior down into your application model services.

INJECTING SERVICES INTO VIEW TEMPLATES

Injecting dependencies into the constructor is recommended, but what if you don’t *have* a constructor? In particular, how do you go about injecting services into a Razor view template when you don’t have control over how the template is constructed?

Imagine you have a simple service, `HtmlGenerator`, to help you generate HTML in your view templates. The question is, how do you pass this service to your view templates, assuming you’ve already registered it with the DI container?

One option is to inject the `HtmlGenerator` into your Razor Page using constructor injection and expose the service as a property on your `PageModel`, as you saw in chapter 7. This will often be the easiest approach, but in some cases, you might not want to have references to the `HtmlGenerator` service in your `PageModel` at all. In those cases, you can directly inject `HtmlGenerator` into your view templates.

> **NOTE** Some people take offense to injecting services into views in this way. You definitely shouldn’t be injecting services related to business logic into your views, but I think it makes sense for services that are related to HTML generation.

You can inject a service into a Razor template with the `@inject` directive, by providing the type to inject and a name for the injected service in the template.

Listing 10.16 Injecting a service into a Razor view template with `@inject`

```
@inject HtmlGenerator htmlHelper   #A
<h1>The page title</h1>
<footer>
    @htmlHelper.Copyright()        #B
</footer>
```

#A Injects an instance of HtmlGenerator into the view, named htmlHelper
#B Uses the injected service by calling the htmlHelper instance

Injecting services directly into views can be a useful way of exposing UI-related services to your view templates without having to take a dependency on the service in your `PageModel`. You probably won’t find you need to rely on it too much, but it’s a useful tool to have.

That pretty much covers registering and using dependencies, but there’s one important aspect I’ve only vaguely touched on: lifetimes, or, when does the container create a new instance of a service? Understanding lifetimes is crucial to working with DI containers, so it’s important to pay close attention to them when registering your services with the container.

10.3 Understanding lifetimes: when are services created?

Whenever the DI container is asked for a particular registered service, for example an instance of `IMessageSender`, it can do one of two things:

- Create and return a new instance of the service
- Return an existing instance of the service

The *lifetime* of a service controls the behavior of the DI container with respect to these two options. You define the lifetime of a service during DI service registration. This dictates when a DI container will reuse an existing instance of the service to fulfill service dependencies, and when it will create a new one.

> **DEFINITION** The lifetime of a service is how long an instance of a service should *live* in a container before it creates a new instance.

It's important to get your head around the implications for the different lifetimes used in ASP.NET Core, so this section looks at each available lifetime option and when you should use it. In particular, you'll see how the lifetime affects how often the DI container creates new objects. In section 10.3.4, I show you a pattern of lifetimes to look out for, where a short-lifetime dependency is "captured" by a long-lifetime dependency. This can cause some hard-to-debug issues, so it's important to bear in mind when configuring your app!

In ASP.NET Core, you can specify three different lifetimes when registering a service with the built-in container:

- *Transient*—Every time a service is requested, a new instance is created. This means you can potentially have different instances of the same class within the same dependency graph.
- *Scoped*—Within a *scope*, all requests for a service will give you the same object. For different scopes you'll get different objects. In ASP.NET Core, each web request gets its own scope.
- *Singleton*—You'll always get the same instance of the service, no matter which scope.

> **NOTE** These concepts align well with most other DI containers, but the terminology often differs. If you're familiar with a third-party DI container, be sure you understand how the lifetime concepts align with the built-in ASP.NET Core DI container.

To illustrate the behavior of each lifetime, in this section, I'll use a simple representative example. Imagine you have `DataContext`, which has a connection to a database, as shown in listing 10.17. It has a single property, `RowCount`, which displays the number of rows in the `Users` table of a database. For the purposes of this example, we emulate calling the database, by setting the number of rows in the constructor, so you will get the same value every time you call `RowCount` on a given `DataContext` instance. *Different* instances of `DataContext` will return a *different* `RowCount` value.

Listing 10.17 `DataContext` generating a random `RowCount` in its constructor

```
public class DataContext
{
    static readonly Random _rand = new Random();
    public DataContext()
    {
        RowCount = _rand.Next(1, 1_000_000_000);     #A
    }

    public int RowCount { get; }                     #B
}
```

#A Generates a random number between 1 and 1,000,000,000
#B Read-only property set in the constructor, so always returns the same value.

You also have a `Repository` class that has a dependency on the `DataContext`, as shown in the next listing. This also exposes a `RowCount` property, but this property delegates the call to its instance of `DataContext`. Whatever value `DataContext` was created with, the `Repository` will display the same value.

Listing 10.18 Repository service that depends on an instance of `DataContext`

```
public class Repository
{
    private readonly DataContext _dataContext;    #A
    public Repository(DataContext dataContext)    #A
    {                                                 #A
        _dataContext = dataContext;                   #A
    }                                                 #A
    public int RowCount => _dataContext.RowCount;     #B
}
```

#A An instance of DataContext is provided using DI.
#B RowCount returns the same value as the current instance of DataContext.

Finally, you have the Razor Page `RowCountModel`, which takes a dependency on both `Repository` and on `DataContext` directly. When the Razor Page activator creates an instance of `RowCountModel`, the DI container injects an instance of `DataContext` and an instance of `Repository`. To create `Repository`, it also creates a second instance of `DataContext`. Over the course of two requests, a total of *four* instances of `DataContext` will be required, as shown in figure 10.12.

`RowCountModel` records the value of `RowCount` returned from both `Repository` and `DataContext` as properties on the `PageModel`. These are then rendered using a Razor template (not shown).

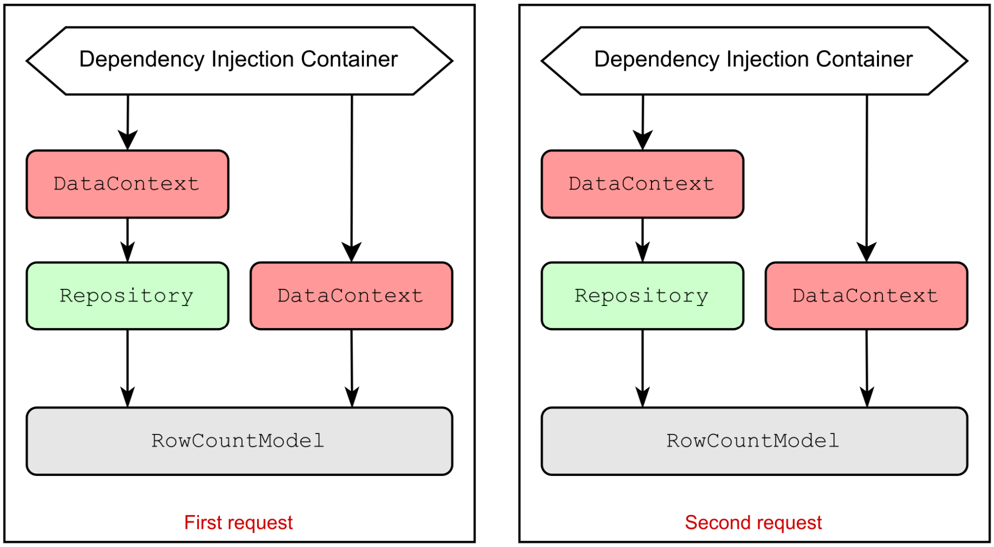
Listing 10.19 `RowCountModel` depends on `DataContext` and `Repository`

```
public class RowCountModel : PageModel
{
    private readonly Repository _repository;       #A
    private readonly DataContext _dataContext;     #A
    public RowCountPageModel(                      #A
        Repository repository,                     #A
        DataContext dataContext)                   #A
    {                                              #A
        _repository = repository;                  #A
        _dataContext = dataContext;                #A
    }                                              #A

     public void OnGet()
    {
        DataContextCount = _dataContext.RowCount;    #B
        RepositoryCount = _repository.RowCount;      #B
    }

    public int DataContextCount { get; set ;}      #C
    public int RepositoryCount { get; set ;}       #C
}
```

#A DataContext and Repository are passed in using DI.
#B When invoked, the page handler retrieves and records RowCount from both dependencies.
#C The counts are exposed on the PageModel and are rendered to HTML in the Razor view.

The purpose of this example is to explore the relationship between the four `DataContext` instances, depending on the lifetimes you use to register the services with the container. I'm generating a random number in `DataContext` as a way of uniquely identifying a `DataContext` instance, but you can think of this as being a point-in-time snapshot of the number of users logged in to your site, for example, or the amount of stock in a warehouse.

I'll start with the shortest-lived lifetime, "transient", move on to the common "scoped" lifetime, and then take a look at "singletons". Finally, I'll show an important trap you should be on the lookout for when registering services in your own apps.



For each request, two instances of DataContext are required to build the RowCountModel instance.

A total of four DataContext instances are required for two requests.

Figure 10.12 The DI Container uses two instances of `DataContext` for each request. Depending on the lifetime with which the `DataContext` type is registered, the container might create one, two, or four different instances of `DataContext`.

10.3.1 Transient: everyone is unique

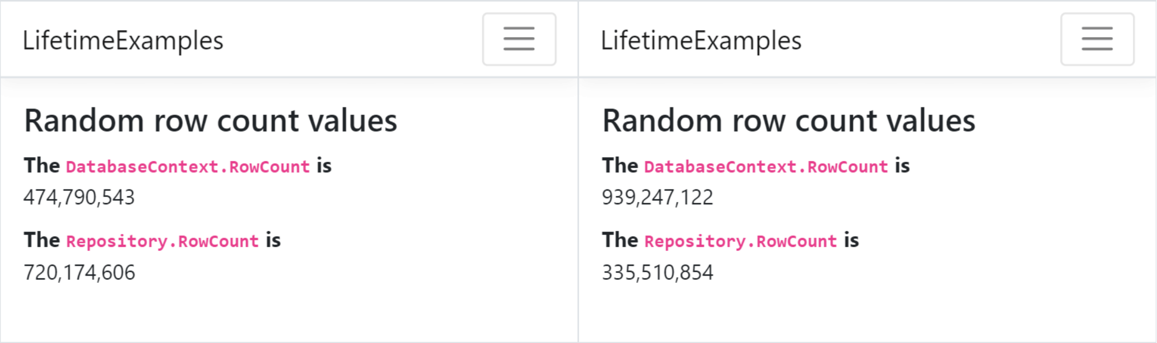
In the ASP.NET Core DI container, transient services are always created new, whenever they’re needed to fulfill a dependency. You can register your services using the `AddTransient` extension methods:

```
services.AddTransient<DataContext>();
services.AddTransient<Repository>();
```

When registered in this way, every time a dependency is required, the container will create a new one. This applies both *between* requests but also *within* requests; the `DataContext` injected into the Repository will be a different instance to the one injected into the `RowCountModel`.

> **NOTE** **Transient dependencies can result in different instances of the same type within a single dependency graph.**

Figure 10.13 shows the results you get from two consecutive requests when you use the transient lifetime for both services. Note that, by default, Razor Page and API controller instances are also transient and are always created anew.



Random row count values

The DatabaseContext.RowCount is
474,790,543

The Repository.RowCount is
720,174,606

Random row count values

The DatabaseContext.RowCount is
939,247,122

The Repository.RowCount is
335,510,854

Figure 10.13 When registered using the transient lifetime, all four `DataContext` objects are different. That can be seen by the four different numbers displayed over the course of two requests.

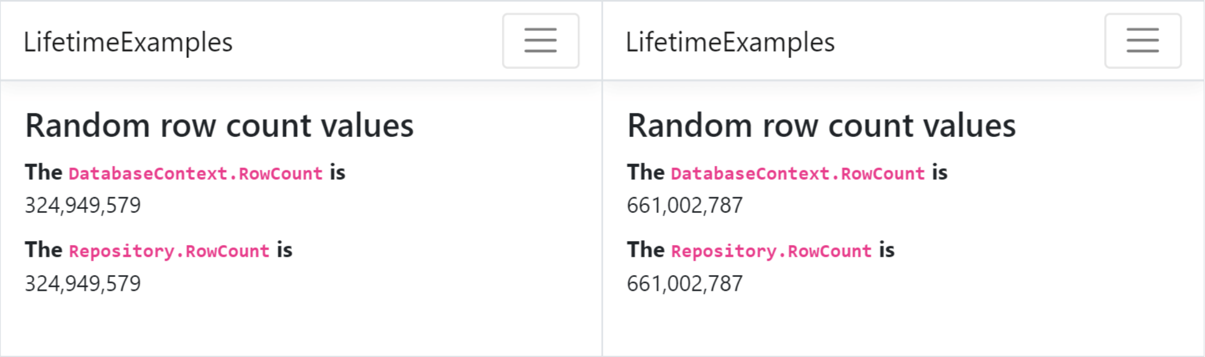
Transient lifetimes can result in a lot of objects being created, so they make the most sense for lightweight services with little or no state. It's equivalent to calling `new` every time you need a new object, so bear that in mind when using it. You probably won't use the transient lifetime too often; the majority of your services will probably be *scoped* instead.

10.3.2 Scoped: let's stick together

The scoped lifetime states that a *single* instance of an object will be used *within* a given scope, but a *different* instance will be used *between different scopes*. In ASP.NET Core, a scope maps to a request, so within a single request the container will use the same object to fulfill all dependencies.

For the row count example, that means that, within a single request (a single scope), the same `DataContext` will be used throughout the dependency graph. The `DataContext` injected into the `Repository` will be the same instance as that injected into `RowCountModel`.

In the next request, you'll be in a different scope, so the container will create a new instance of `DataContext`, as shown in figure 10.14. A different instance means a different `RowCount` for each request, as you can see.



LifetimeExamples

Random row count values

The DatabaseContext.RowCount is

324,949,579

The Repository.RowCount is

324,949,579

LifetimeExamples

Random row count values

The DatabaseContext.RowCount is

661,002,787

The Repository.RowCount is

661,002,787

Figure 10.14 Scoped dependencies use the same instance of `DataContext` within a single request, but a new instance for a separate request. Consequently, the `RowCount`s are identical within a request.

You can register dependencies as scoped using the `AddScoped` extension methods. In this example, I registered `DataContext` as scoped and left `Repository` as transient, but you'd get the same results in this case if they were both scoped:

```
services.AddScoped<DataContext>();
```

Due to the nature of web requests, you'll often find services registered as scoped dependencies in ASP.NET Core. Database contexts and authentication services are common examples of services that should be scoped to a request—anything that you want to share across your services *within* a single request, but that needs to change *between* requests.

Generally speaking, you'll find a lot of services registered using the scoped lifetime—especially anything that uses a database or is dependent on a specific request. But some services don't need to change between requests, such as a service that calculates the area of a circle, or that returns the current time in different time zones. For these, a singleton lifetime might be more appropriate.

10.3.3 Singleton: there can be only one

The singleton is a pattern that came before dependency injection; the DI container provides a robust and easy-to-use implementation of it. The singleton is conceptually simple: an instance of the service is created when it's first needed (or during registration, as in section 10.2.3) and that's it: you'll always get the same instance injected into your services.

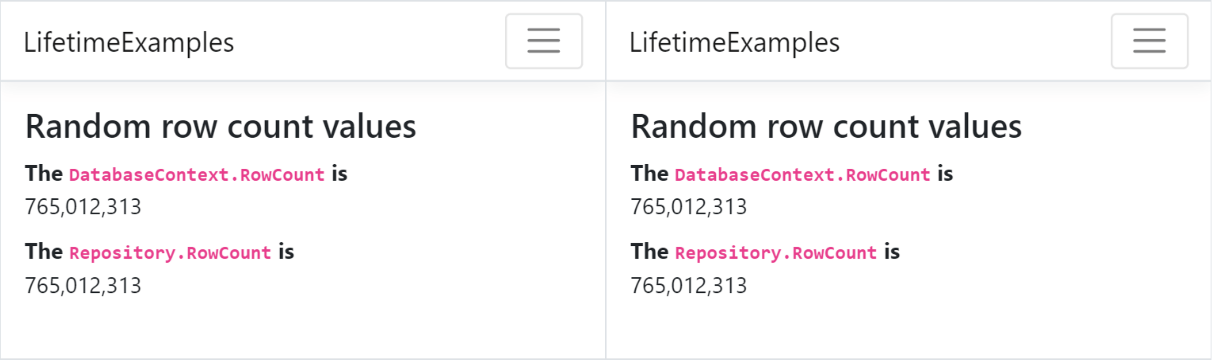
The singleton pattern is particularly useful for objects that are either expensive to create, contain data that must be shared across requests, or that don't hold state. The latter two points are important—any service registered as a singleton should be thread-safe.

> **WARNING** Singleton services must be thread-safe in a web application, as they'll typically be used by multiple threads during concurrent requests.

Let's consider what using singletons means for the row count example. I can update the registration of `DataContext` to be a singleton in `ConfigureServices`, using

```
services.AddSingleton<DataContext>();
```

We then call the `RowCountModel` Razor Page twice and observe the results shown in figure 10.15. You can see that every instance has returned the same value, indicating that all four instances of `DataContext` are the same single instance.



Random row count values

The `DatabaseContext.RowCount` is
765,012,313

The `Repository.RowCount` is
765,012,313

Random row count values

The `DatabaseContext.RowCount` is
765,012,313

The `Repository.RowCount` is
765,012,313

Figure 10.15 Any service registered as a singleton will always return the same instance. Consequently, all the calls to `RowCount` return the same value, both within a request and between requests.

Singletons are convenient for objects that need to be shared or that are immutable and expensive to create. A caching service should be a singleton, as all requests need to share it. It must be thread-safe though. Similarly, you might register a settings object loaded from a remote server as a singleton, if you loaded the settings once at startup and reused them through the lifetime of your app.

On the face of it, choosing a lifetime for a service might not seem too tricky, but there's an important "gotcha" that can come back to bite you in subtle ways, as you'll see shortly.

10.3.4 Keeping an eye out for captured dependencies

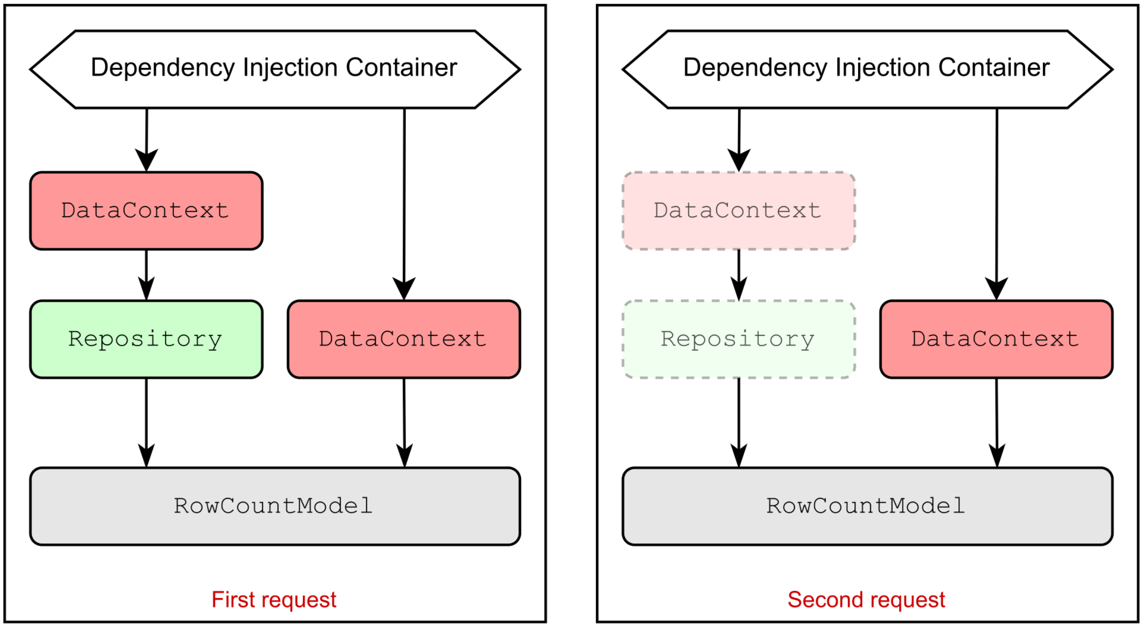
Imagine you're configuring the lifetime for the `DataContext` and `Repository` examples. You think about the suggestions I've provided and decide on the following lifetimes:

- `DataContext`—*Scoped*, as it should be shared for a single request
- `Repository`—*Singleton*, as it has no state of its own and is thread-safe, so why not?

> **WARNING** This lifetime configuration is to explore a bug—don't use it in your code or you'll experience a similar problem!

Unfortunately, you've created a *captured dependency* because you're injecting a *scoped* object, `DataContext`, into a *singleton*, `Repository`. As it's a singleton, the same

`Repository` instance is used throughout the lifetime of the app, so the `DataContext` that was injected into it will *also* hang around, *even though a new one should be used with every request*. Figure 10.16 shows this scenario, where a new instance of `DataContext` is created for each scope, but the instance inside `Repository` hangs around for the lifetime of the app.



As the repository has been registered as a singleton, the DataContext it uses will act also as a singleton, even though it is registered as scoped.

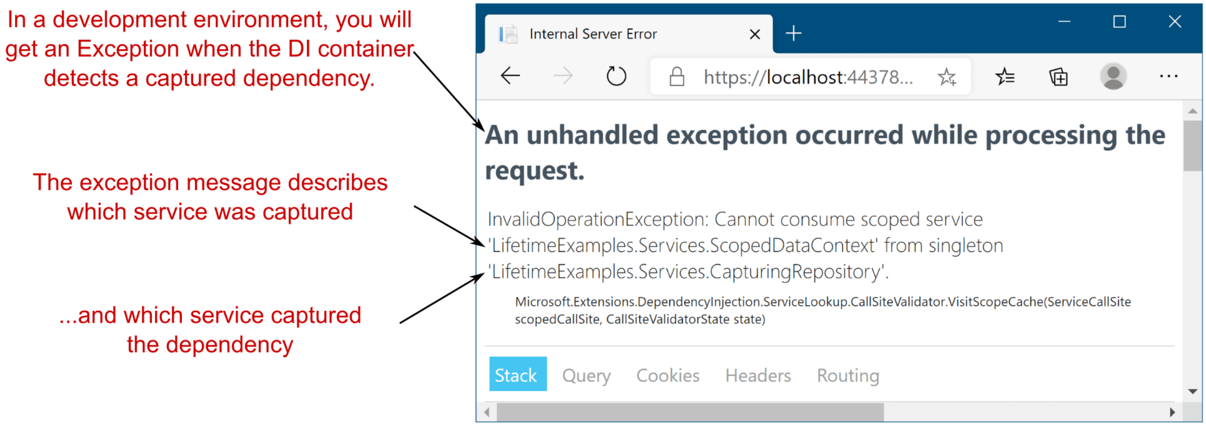
The DataContext dependency has been captured by the repository, breaking the scoped lifetime.

Figure 10.16 `DataContext` is registered as a scoped dependency, but `Repository` is a singleton. Even though you expect a new `DataContext` for every request, `Repository` *captures* the injected `DataContext` and causes it to be reused for the lifetime of the app.

Captured dependencies can cause subtle bugs that are hard to root out, so you should always keep an eye out for them. These captured dependencies are relatively easy to introduce, so always think carefully when registering a singleton service.

> **WARNING** A service should only use dependencies with a lifetime longer than or equal to the lifetime of the service. A service registered as a singleton can only safely use singleton dependencies. A service registered as scoped can safely use scoped or singleton dependencies. A transient service can use dependencies with any lifetime.

At this point, I should mention that there’s one glimmer of hope in this cautionary tale. ASP.NET Core automatically checks for these kinds of captured dependencies and will throw an exception on application startup if it detects them, as shown in figure 10.17.



An unhandled exception occurred while processing the request.

InvalidOperationException: Cannot consume scoped service 'LifetimeExamples.Services.ScopedDataContext' from singleton 'LifetimeExamples.Services.CapturingRepository'.

Microsoft.Extensions.DependencyInjection.ServiceLookup.CallSiteValidator.VisitScopeCache(ServiceCallSite scopedCallSite, CallSiteValidatorState state)

In a development environment, you will get an Exception when the DI container detects a captured dependency.

The exception message describes which service was captured

...and which service captured the dependency

Figure 10.17 When `ValidateScopes` is enabled, the DI container will throw an exception when it creates a service with a captured dependency. By default, this check is only enabled for development environments.

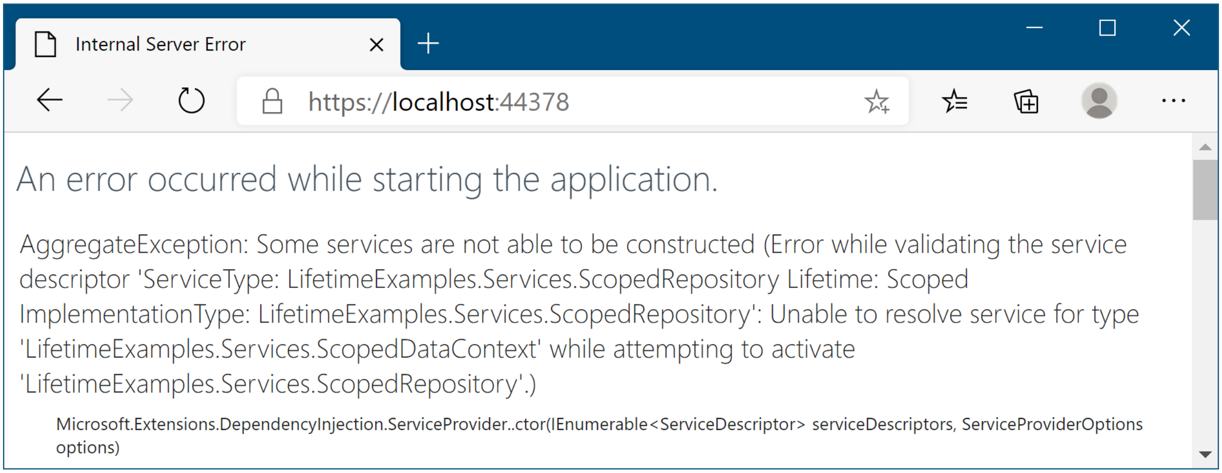
This scope validation check has a performance impact, so by default it’s only enabled when your app is running in a development environment, but it should help you catch most issues of this kind. You can enable or disable this check regardless of environment by setting the `ValidateScopes` option when creating your `HostBuilder` in Program.cs, as shown in the following listing.

Listing 10.20 Setting the `ValidateScopes` property to always validate scopes

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)                          #A
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            })
            .UseDefaultServiceProvider(options =>          #B
            {
                options.ValidateScopes = true;             #C
                options.ValidateOnBuild = true;            #D
            });
}
```

#A The default builder sets ValidateScopes to validate only in development environments.
#B You can override the validation check with the UseDefaultServiceProvider extension.
#C Setting to true will validate scopes in all environments. This has performance implications.
#D ValidateOnBuild checks that every registered service has all its dependencies registered.

Listing 10.20 shows another setting you can enable, `ValidateOnBuild`, that goes one step further. When enabled, the DI container checks on application startup that it has dependencies registered for every service it needs to build. If it doesn't, it throws an exception, as shown in figure 10.18, letting you know about the misconfiguration. This also has a performance impact, so it's only enabled in development environments by default, but it's very useful for pointing out any missed service registrations![47]



An error occurred while starting the application.

AggregateException: Some services are not able to be constructed (Error while validating the service descriptor 'ServiceType: LifetimeExamples.Services.ScopedRepository Lifetime: Scoped ImplementationType: LifetimeExamples.Services.ScopedRepository': Unable to resolve service for type 'LifetimeExamples.Services.ScopedDataContext' while attempting to activate 'LifetimeExamples.Services.ScopedRepository'.)

Microsoft.Extensions.DependencyInjection.ServiceProvider..ctor(IEnumerable<ServiceDescriptor> serviceDescriptors, ServiceProviderOptions options)

Figure 10.17 When `ValidateOnBuild` is enabled, the DI container will check on app start up that it can create all of the registered services. If it finds a service it can't create, it throws an exception. By default, this check is only enabled for development environments.

With that, you've reached the end of this introduction to DI in ASP.NET Core. You now know how to add framework services to your app using `Add*` extension methods like `AddRazorPages()`, as well as how to register your own services with the DI container. Hopefully, that will help you keep your code loosely coupled and easy to manage.

In the next chapter, we'll look at the ASP.NET Core configuration model. You'll see how to load settings from a file at runtime, how to store sensitive settings safely, and how to make

[47] **Unfortunately, the container can't catch everything. For a list of caveats and exceptions, see this post: https://andrewlock.net/new-in-asp-net-core-3-service-provider-validation/.**

your application behave differently depending on which machine it's running on. We'll even use a bit of DI; it gets everywhere in ASP.NET Core!

10.4 Summary

- Dependency injection is baked into the ASP.NET Core framework. You need to ensure your application adds all the framework's dependencies in `Startup` otherwise you will get exceptions at runtime when the DI container can't find the required services.
- The dependency graph is the set of objects that must be created in order to create a specific requested "root" object. The DI container handles creating all these dependencies for you.
- You should aim to use explicit dependencies over implicit dependencies in most cases. ASP.NET Core uses constructor arguments to declare explicit dependencies.
- When discussing DI, the term *service* is used to describe any class or interface registered with the container.
- You register services with a DI container so it knows which implementation to use for each requested service. This typically takes the form of, "for interface X, use implementation Y."
- The DI or IoC container is responsible for creating instances of services. It knows how to construct an instance of a service by creating all the service's dependencies and passing these in the service constructor.
- The default built-in container only supports constructor injection. If you require other forms of DI, such as property injection, you can use a third-party container.
- You must register services with the container by calling `Add*` extension methods on `IServiceCollection` in `ConfigureServices` in `Startup`. If you forget to register a service that's used by the framework or in your own code, you'll get an `InvalidOperationException` at runtime.
- When registering your services, you describe three things: the service type, the implementation type, and the lifetime. The service type defines which class or interface will be requested as a dependency. The implementation type is the class the container should create to fulfil the dependency. The lifetime is how long an instance of the service should be used for.
- You can register a service using generic methods if the class is concrete and all its constructor arguments are registered with the container or have default values.
- You can provide an instance of a service during registration, which will register that instance as a singleton. This can be useful when you already have an instance of the service available.
- You can provide a lambda factory function that describes how to create an instance of a service with any lifetime you choose. You can use this approach when your services depend on other services, which are only accessible once your application is running.
- Avoid calling `GetService()` in your factory functions if possible. Instead, favor constructor injection—it's more performant, as well as being simpler to reason about.

- You can register multiple implementations for a service. You can then inject `IEnumerable<T>` to get access to all the implementations at runtime.
- If you inject a single instance of a multiple-registered service, the container injects the last implementation registered.
- You can use the `TryAdd*` extension methods to ensure that an implementation is only registered if no other implementation of the service has been registered. This can be useful for library authors to add “default” services while still allowing consumers to override the registered services.
- You define the lifetime of a service during DI service registration. This dictates when a DI container will reuse an existing instance of the service to fulfil service dependencies and when it will create a new one.
- The transient lifetime means that every single time a service is requested, a new instance is created.
- The scoped lifetime means that within a scope, all requests for a service will give you the same object. For different scopes, you’ll get different objects. In ASP.NET Core, each web request gets its own scope.
- You’ll always get the same instance of a singleton service, no matter which scope.
- A service should only use dependencies with a lifetime longer than or equal to the lifetime of the service.

11

Configuring an ASP.NET Core application

This chapter covers

- **Loading settings from multiple configuration providers**
- **Storing sensitive settings safely**
- **Using strongly typed settings objects**
- **Using different settings in different hosting environments**

In part 1 of this book, you learned the basics of getting an ASP.NET Core app up and running and how to use the MVC design pattern to create a traditional web app or a Web API. Once you start building real applications, you will quickly find that you want to tweak various settings at deploy time, without necessarily having to recompile your application. This chapter looks at how you can achieve this in ASP.NET Core using configuration.

I know. Configuration sounds boring, right? But I have to confess, the configuration model is one of my favorite parts of ASP.NET Core. It's so easy to use, and so much more elegant than the previous version of ASP.NET. In section 11.2, you'll learn how to load values from a plethora of sources—JSON files, environment variables, and command-line arguments—and combine them into a unified configuration object.

On top of that, ASP.NET Core brings the ability to easily bind this configuration to strongly typed *options* objects. These are simple, POCO classes that are populated from the configuration object, which you can inject into your services, as you'll see in section 11.3. This lets you nicely encapsulate settings for different features in your app.

In the final section of this chapter, you'll learn about the ASP.NET Core *hosting environments*. You often want your app to run differently in different situations such as when running on your developer machine compared to when you deploy it to a production server.

These different situations are known as *environments*. By letting the app know in which environment it’s running, it can load a different configuration and vary its behavior accordingly.

Before we get to that, let’s go back to basics: what is configuration, why do we need it, and how does ASP.NET Core handle these requirements?

11.1 Introducing the ASP.NET Core configuration model

In this section I provide a brief description of what we mean by configuration and what you can use it for in ASP.NET Core applications. Configuration is the set of external parameters provided to an application that controls the application’s behavior in some way. It typically consists of a mixture of *settings* and *secrets* that the application will load at runtime.

> **DEFINITION** A *setting* is any value that changes the behavior of your application. A *secret* is a special type of setting that contains sensitive data, such as a password, an API key for a third-party service, or a connection string.

The obvious question before we get started is to consider why you need app configuration, and what sort of things you need to configure. You should normally move anything that you can consider a setting or a secret out of your application code. That way, you can easily change these values at deploy time, without having to recompile your application.

You might, for example, have an application that shows the locations of your brick-and-mortar stores. You could have a setting for the connection string to the database in which you store the details of the stores, but also settings such as the default location to display on a map, the default zoom level to use, and the API key for accessing the Google Maps API, as shown in figure 11.1. Storing these settings and secrets outside of your compiled code is good practice as it makes it easy to tweak them without having to recompile your code.

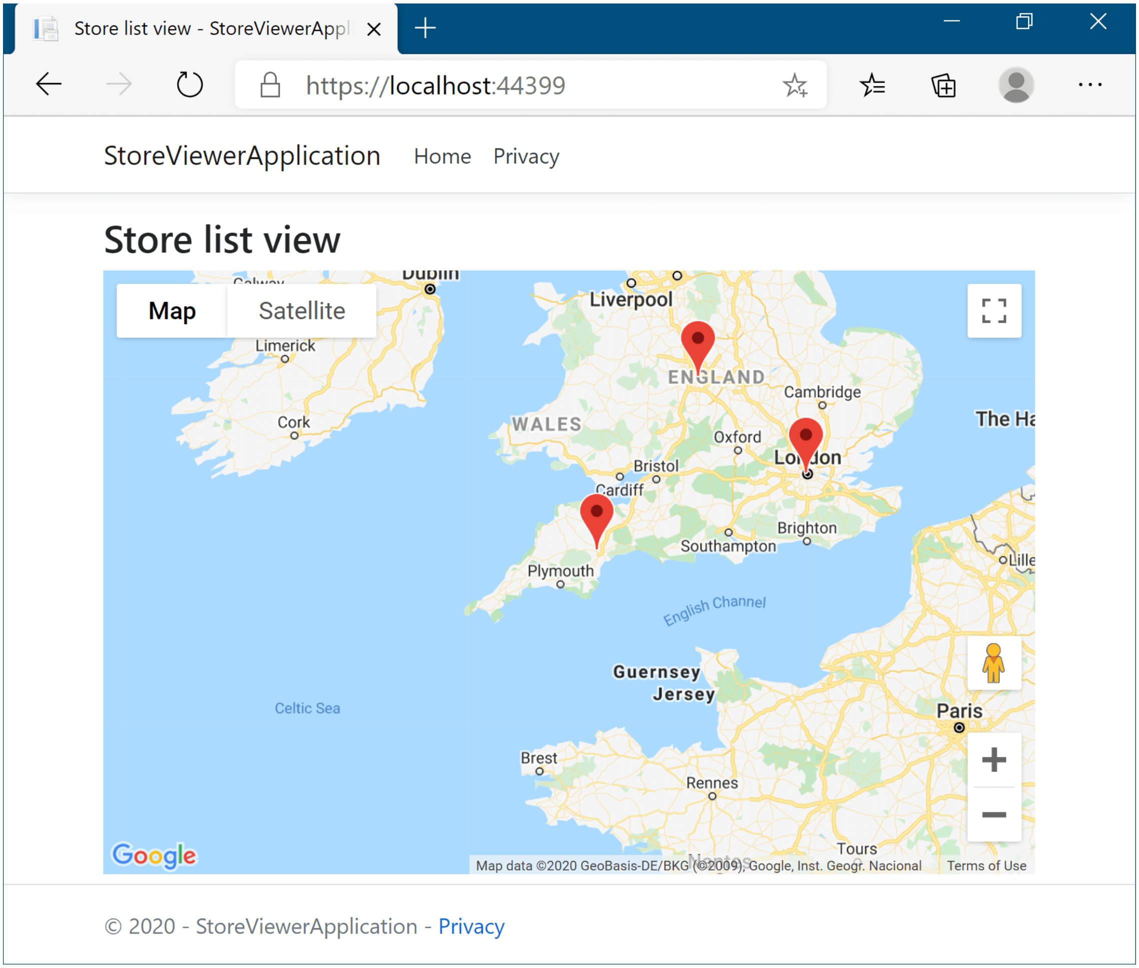


Figure 11.1 You can store the default map location, zoom level, and mapping API Key in configuration and load them at runtime. It's important to keep secrets like API keys in configuration and out of your code.

There's also a security aspect to this; you don't want to hardcode secret values like API keys or passwords into your code, where they could be committed to source control and made publicly available. Even values embedded in your compiled application can be extracted, so it's best to externalize them whenever possible.

Virtually every web framework provides a mechanism for loading configuration and the previous version of ASP.NET was no different. It used the `<appsettings>` element in a web.config file to store key-value configuration pairs. At runtime, you'd use the static (*wince*) `ConfigurationManager` to load the value for a given key from the file. You could do more advanced things using custom configuration sections, but this was painful, and so was rarely used, in my experience.

ASP.NET Core gives you a totally revamped experience. At the most basic level, you're still specifying key-value pairs as strings, but instead of getting those values from a single file, you can now load them from multiple sources. You can load values from files, but they can now be any format you like: JSON, XML, YAML, and so on. On top of that, you can load values from environment variables, from command-line arguments, from a database, or from a remote service. Or you could create your own custom *configuration provider*.

> **DEFINITION** ASP.NET Core uses *configuration providers* to load key-value pairs from a variety of sources. Applications can use many different configuration providers.

The ASP.NET Core configuration model also has the concept of *overriding* settings. Each configuration provider can define its own settings, or it can overwrite settings from a previous provider. You'll see this incredibly-useful feature in action in section 11.3.

ASP.NET Core makes it simple to bind these key-value pairs, which are defined as `strings`, to POCO-setting classes you define in your code. This model of strongly typed configuration makes it easy to logically group settings around a given feature and lends itself well to unit testing.

Before we get into the details of loading configuration from providers, we'll take a step back and look at *where* this process happens—inside `HostBuilder`. For ASP.NET Core 3.1 apps built using the default templates, that's invariably inside the `Host.CreateDefaultBuilder()` method in Program.cs.

11.2 Configuring your application with CreateDefaultBuilder

As you saw in chapter 2, the default templates in ASP.NET Core 3.1 use the `CreateDefaultBuilder` method. This is an opinionated helper method that sets up a number of defaults for your app. In this section we'll look inside this method to see all the things it configures, and what they're used for.

Listing 11.1 Using `CreateDefaultBuilder` to set up configuration

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();                #A
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)                  #B
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

#A The entry point for your application creates an IHostBuilder, builds an IHost, and calls Run.
#B CreateDefaultBuilder sets up a number of defaults, including configuration.

In chapter 2, I glossed over this method, as you will only rarely need to change it for simple apps. But as your application grows, and if you want to change how configuration is loaded for your application, you may find you need to break it apart.

This listing shows an overview of the `CreateDefaultBuilder` method, so you can see how `HostBuilder` is constructed.

Listing 11.2 The `WebHost.CreateDefaultBuilder` method

```
public static IHostBuilder CreateDefaultBuilder(string[] args)
{
  var builder = new HostBuilder()                               #A
    .UseContentRoot(Directory.GetCurrentDirectory())            #B
    .ConfigureHostConfiguration(config =>                       #C
    {                                                           #C
        // Configuration provider setup                         #C
    })                                                          #C
    .ConfigureAppConfiguration((hostingContext, config) =>      #D
    {                                                           #D
        // Configuration provider setup                         #D
    })                                                          #D
    .ConfigureLogging((hostingContext, logging) =>              #E
    {                                                           #E
        logging.AddConfiguration(                               #E
          hostingContext.Configuration.GetSection("Logging"));  #E
        logging.AddConsole();                                   #E
        logging.AddDebug();                                     #E
    })                                                          #E
    .UseDefaultServiceProvider((context, options) =>            #F
    {                                                           #F
        var isDevelopment = context.HostingEnvironment          #F
                                   .IsDevelopment();            #F
        options.ValidateScopes = isDevelopment;                 #F
        options.ValidateOnBuild = isDevelopment;                #F
    });                                                         #F

  return builder;                                               #G
}
```

#A Creating an instance of HostBuilder
#B The content root defines the directory where configuration files can be found.
#C Configures hosting settings such as determining the hosting environment
#D Configures application settings, the topic of this chapter
#E Sets up the logging infrastructure
#F Configures the DI container, optionally enabling verification settings
#G Returns HostBuilder for further configuration by calling extra methods before calling Build()

The first method called on `HostBuilder` is `UseContentRoot`. This tells the application in which directory it can find any configuration or view files it will need later. This is typically the folder in which the application is running, hence the call to `GetCurrentDirectory`.

TIP ContentRoot is *not* where you store static files that the browser can access directly—that's the *WebRoot*, typically wwwroot.

The `ConfigureHostingConfiguration()` method is where your application determines which `HostingEnvironment` it's currently running in. The framework looks for environment variables and command line arguments by default, to determine if it's running in a development or production environment. You'll learn more about hosting environments in section 11.5.

`ConfigureLogging` is where you can specify the logging settings for your application. We'll look at logging in detail in chapter 17; for now, it's enough to know that `CreateDefaultBuilder` sets this up for you.

The last method call in `CreateDefaultBuilder`, `UseDefaultServiceProvider`, configures your app to use the built-in DI container. It also sets the `ValidateScopes` and `ValidateOnBuild` options based on the current `HostingEnvironment`. When running the application in the development environment, the app will automatically check for captured dependencies, as you learned about in chapter 10.

The `ConfigureAppConfiguration()` method is the focus of section 11.3. It's where you load the settings and secrets for your app, whether they're in JSON files, environment variables, or command-line arguments. In the next section, you'll see how to use this method to load configuration values from various configuration providers using the ASP.NET Core `ConfigurationBuilder`.

11.3 Building a configuration object for your app

In this section we get into the meat of the configuration system. You'll learn how to load settings from multiple sources, how they're stored internally in ASP.NET Core, and how settings can override other values to give "layers" of configuration. You'll also learn how to store secrets securely, while ensuring they're still available when you run your app.

In section 11.2 you saw how the `CreateDefaultBuilder` method can be used to create an instance of `IHostBuilder`. `IHostBuilder` is responsible for setting up many things about your app, including the configuration system in the `ConfigureAppConfiguration` method. This method is passed an instance of a `ConfigurationBuilder`, which is used to define your app's configuration.

The ASP.NET Core configuration model centers on two main constructs: `ConfigurationBuilder` and `IConfiguration`.

NOTE `ConfigurationBuilder` **describes how to construct the final configuration representation for your app, and** `IConfiguration` **holds the configuration values themselves.**

You describe your configuration setup by adding a number of `IConfigurationProviders` to the `ConfigurationBuilder` in `ConfigureAppConfiguration`. These describe how to load the key-value pairs from a particular source; for example, a JSON file or from environment variables, as shown in figure 11.2. Calling `Build()` on `ConfigurationBuilder` queries each of these providers for the values they contain to create the `IConfigurationRoot` instance.

> **NOTE** Calling `Build()` creates an `IConfigurationRoot` instance, which implements `IConfiguration`. You will generally work with the `IConfiguration` interface in your code.

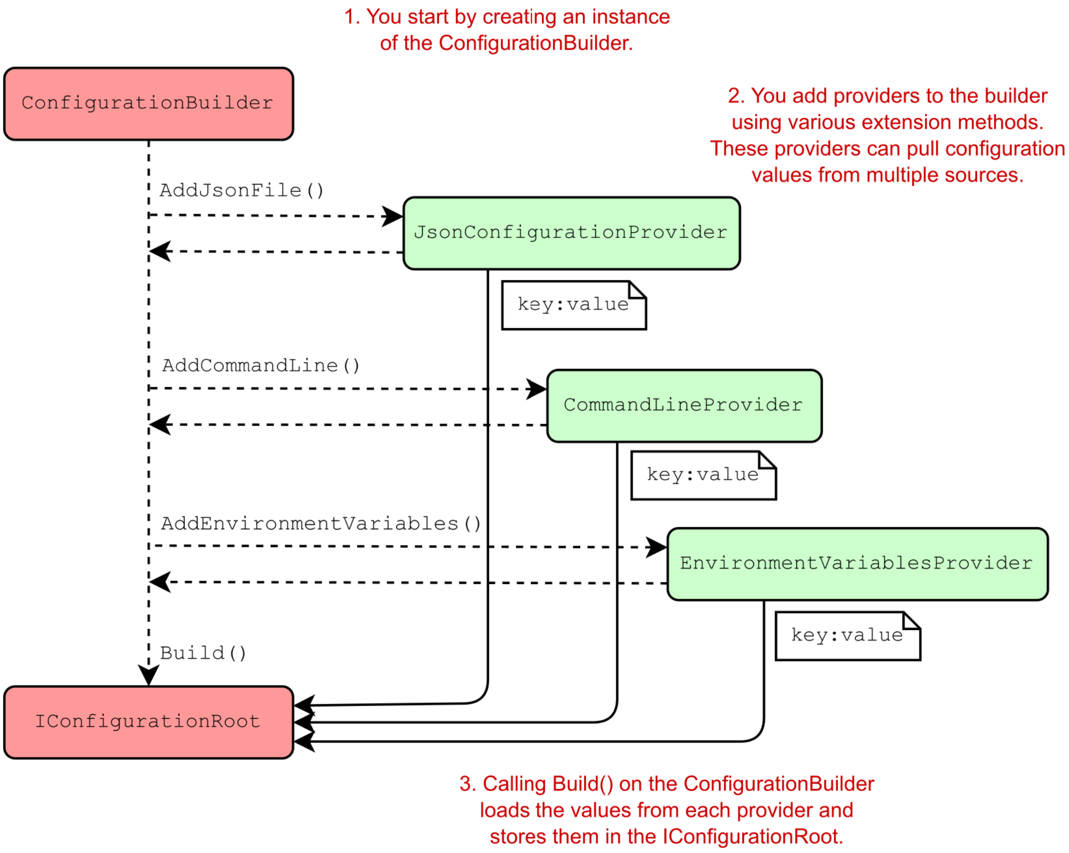

1. You start by creating an instance of the ConfigurationBuilder.
ConfigurationBuilder
2. You add providers to the builder using various extension methods. These providers can pull configuration values from multiple sources.
AddJsonFile()
JsonConfigurationProvider
key:value
AddCommandLine()
CommandLineProvider
key:value
AddEnvironmentVariables()
EnvironmentVariablesProvider
key:value
Build()
IConfigurationRoot
3. Calling Build() on the ConfigurationBuilder loads the values from each provider and stores them in the IConfigurationRoot.

Figure 11.2 Using `ConfigurationBuilder` to create an instance of `IConfiguration`. Configuration providers are added to the builder with extension methods. Calling `Build()` queries each provider to create the `IConfigurationRoot` which implements `IConfiguration`.

ASP.NET Core ships with configuration providers for loading data from common locations:

- JSON files
- XML files
- Environment variables
- Command-line arguments

- INI files

If these don't fit your requirements, you can find a whole host of alternatives on GitHub and NuGet, and it's not difficult to create your own custom provider. For example, you could use the official Azure KeyVault provider NuGet package[48], or the YAML file provider I wrote.[49]

In many cases, the default providers will be sufficient. In particular, most templates start with an appsettings.json file, which contains a variety of settings, depending on the template you choose. This listing below shows the default file generated by the ASP.NET Core 3.1 Web app template without authentication.

Listing 11.3 Default appsettings.json file created by an ASP.NET Core Web template

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

As you can see, this file mostly contains settings to control logging, but you can add additional configuration for your app here too.

> **WARNING** Don't store sensitive values, such as passwords, API keys, or connection strings, in this file. You'll see how to store these securely in section 11.3.3.

Adding your own configuration values involves adding a key-value pair to the JSON. It's a good idea to "namespace" your settings by creating a base object for related settings, as in the `MapSettings` object shown here.

Listing 11.4 Adding additional configuration values to an appsettings.json file

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "MapSettings": {               #A
    "DefaultZoomLevel": 9,       #B
```

[48] The Azure KeyVault provider is available on NuGet: https://www.nuget.org/packages/Microsoft.Extensions.Configuration.AzureKeyVault/.

[49] You can find my YAML provider on GitHub at https://github.com/andrewlock/NetEscapades.Configuration.

```
    "DefaultLocation": {        #C
      "latitude": 50.500,       #C
      "longitude": -4.000       #C
    }
  }
}
```

#A Nest all the configuration under the MapSettings key.
#B Values can be numbers in the JSON file, but they'll be converted to strings when they're read.
#C You can create deeply nested structures to better organize your configuration values.

I've nested the new configurtation inside the `MapSettings` parent key; this creates a "section" which will be useful later when it comes to binding your values to a POCO object. I also nested the `latitude` and `longitude` keys under the `DefaultLocation` key. You can create any structure of values that you like; the configuration provider will read them just fine. Also, you can store them as any data type—numbers, in this case—but be aware that the provider will read and store them internally as strings.

> **TIP** The configuration keys are *not* case-sensitive in your app, so bear that in mind when loading from providers in which the keys *are* case-sensitive.

Now that you have a configuration file, it's time for your app to load it using `ConfigurationBuilder`! For this, return to the `ConfigureAppConfiguration()` method exposed by `HostBuilder` in Program.cs.

11.3.1 Adding a configuration provider in Program.cs

The default templates in ASP.NET Core use the `CreateDefaultBuilder` helper method to bootstrap `HostBuilder` for your app, as you saw in section 11.2. As part of this configuration, the `CreateDefaultBuilder` method calls `ConfigureAppConfiguration` and sets up a number of default configuration providers, which we'll look at in more detail throughout this chapter:

- *JSON file provider*—Loads settings from an optional JSON file called appsettings.json.
- *JSON file provider*—Loads settings from an optional environment-specific JSON file called appsettings.*ENVIRONMENT*.json. I'll show how to use environment-specific files in section 11.5.
- *User Secrets*—Loads secrets that are stored safely during development.
- *Environment variables*—Loads environment variables as configuration variables. Great for storing secrets in production.
- *Command-line arguments*—Uses values passed as arguments when you run your app.

Using the default builder ties you to this default set, but the default builder is optional. If you want to use different configuration providers, you can create your own `HostBuilder` instance instead if you wish. If you take this approach, you'll need to set up everything that `CreateHostBuilder` does: logging, hosting configuration, service provider configuration, as well as your app configuration.

An alternative approach is to add *additional* configuration providers by adding an extra call to ConfigureAppConfiguration, as shown in the following listing. This allows you to add extra providers, on top of those added by CreateHostBuilder. In the listing below, you explicitly clear the default providers, which lets you completely customize where configuration is loaded from, without having to replace the defaults CreateHostBuilder adds for logging and so on.

Listing 11. 5 Loading appsettings.json using a custom WebHostBuilder

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration(AddAppConfiguration)    #A
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });

    public static void AddAppConfiguration(                        #B
        HostBuilderContext hostingContext,                         #B
        IConfigurationBuilder config)                              #B
    {
        config.Sources.Clear();                                    #C
        config.AddJsonFile("appsettings.json", optional: true);    #D
    }
```

#A Adds the configuration setup function to HostBuilder
#B HostBuilder provides a hosting context and an instance of ConfigurationBuilder.
#C Clears the providers configured by default in CreateDefaultBuilder
#D Adds a JSON configuration provider, providing the filename of the configuration file

> **TIP** **In listing 11.5, I extracted the configuration to a static helper method,** AddAppConfiguration**, but you can also provide this inline as a lambda method.**

The HostBuilder creates an ConfigurationBuilder instance before invoking the ConfigureAppConfiguration method. All you need to do is add the configuration providers for your application.

In this example, you've added a single JSON configuration provider by calling the AddJsonFile extension method and providing a filename. You've also set the value of optional to true. This tells the configuration provider to skip over files it can't find at runtime, instead of throwing FileNotFoundException. Note that the extension method just registers the provider at this point, it doesn't try to load the file yet.

And that's it! The HostBuilder instance takes care of calling Build(), which generates IConfiguration, which represents your configuration object. This is then registered with the

DI container, so you can inject it into your classes. You'd commonly inject this into the constructor of your `Startup` class, so you can use it in the `Configure` and `ConfigureServices` methods:

```
public class Startup
{
    public Startup(IConfiguration config)
    {
        Configuration = config
    }
    public IConfiguration Configuration { get; }
}
```

> **NOTE** **The `ConfigurationBuilder` creates an `IConfigurationRoot` instance, which implements `IConfiguration`. This is registered, as an `IConfiguration` in the DI container, *not* an `IConfigurationRoot`. `IConfiguration` is one of the few things that you can inject into the `Startup` constructor.**

At this point, at the end of the `Startup` constructor, you have a fully loaded configuration object! But what can you do with it? The `IConfiguration` stores configuration as a set of key-value `string` pairs. You can access any value by its key, using standard dictionary syntax. For example, you could use

```
var zoomLevel = Configuration["MapSettings:DefaultZoomLevel"];
```

to retrieve the configured zoom level for your application. Note that I used a colon "`:`" to designate a separate section. Similarly, to retrieve the `latitude` key, you could use

```
Configuration["MapSettings:DefaultLocation:Latitude"];
```

> **NOTE** **If the requested configuration key does not exist, you will get a `null` value.**

You can also grab a whole section of the configuration using the `GetSection(section)` method, which returns an `IConfigurationSection`, which implements `IConfiguration`. This grabs a chunk of the configuration and resets the namespace. Another way of getting the latitude key would be

```
Configuration.GetSection("MapSettings")["DefaultLocation:Latitude"];
```

Accessing setting values like this is useful in the `ConfigureServices` and `Configure` methods of `Startup`, when you're defining your application. When setting up your application to connect to a database, for example, you'll often load a connection string from the `Configuration` object (you'll see a concrete example of this in the next chapter, when we look at Entity Framework Core).

If you need to access the configuration object like this from classes other than `Startup`, you can use DI to take it as a dependency in your service's constructor. But accessing configuration using `string` keys like this isn't particularly convenient; you should try to use strongly typed configuration instead, as you'll see in section 11.4.

So far, this is probably all feeling a bit convoluted and run-of-the-mill to load settings from a JSON file, and I'll grant you, it is. Where the ASP.NET Core configuration system shines is when you have multiple providers.

11.3.2 Using multiple providers to override configuration values

You've seen that ASP.NET Core uses the builder pattern to construct the configuration object, but so far, you've only configured a single provider. When you add providers, it's important to consider the order in which you add them, as that defines the order in which the configuration values are added to the underlying dictionary. Configuration values from later providers will overwrite values with the same key from earlier providers.

> **NOTE** **This bears repeating: the order you add configuration providers to `ConfigurationBuilder` is important. Later configuration providers can overwrite the values of earlier providers.**

Think of the configuration providers as adding "layers" of configuration values to a stack, where each layer may overlap with some or all of the layers below, as shown in figure 11.3. When you call `Build()`, `ConfigurationBuilder` collapses these layers down into one, to create the final set of configuration values stored in `IConfiguration`.

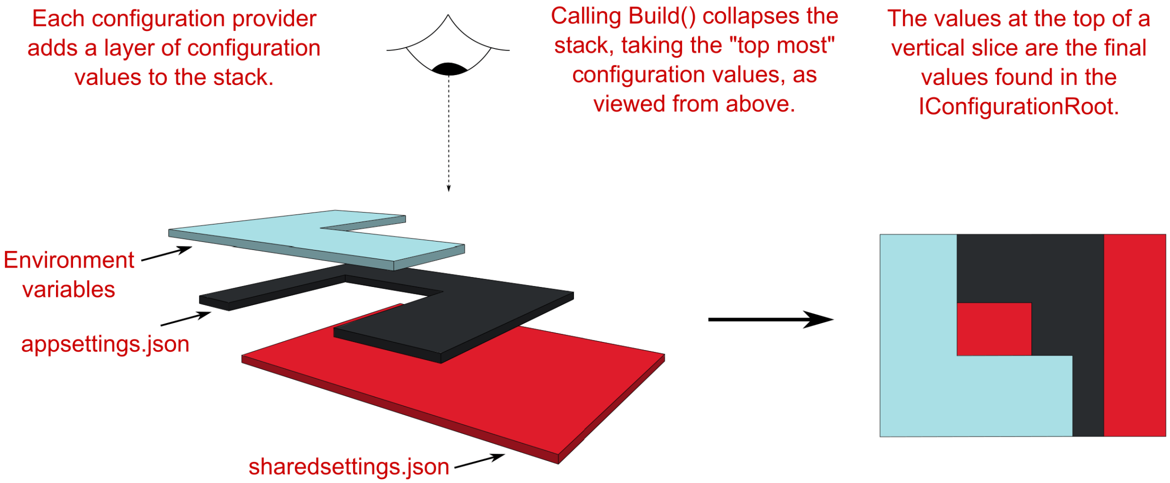

Each configuration provider adds a layer of configuration values to the stack.
Calling Build() collapses the stack, taking the "top most" configuration values, as viewed from above.
The values at the top of a vertical slice are the final values found in the IConfigurationRoot.
Environment variables
appsettings.json
sharedsettings.json

Figure 11.3 Each configuration provider adds a "layer" of values to `ConfigurationBuilder`. Calling `Build()` collapses that configuration. Later providers will overwrite configuration values with the same key as earlier providers.

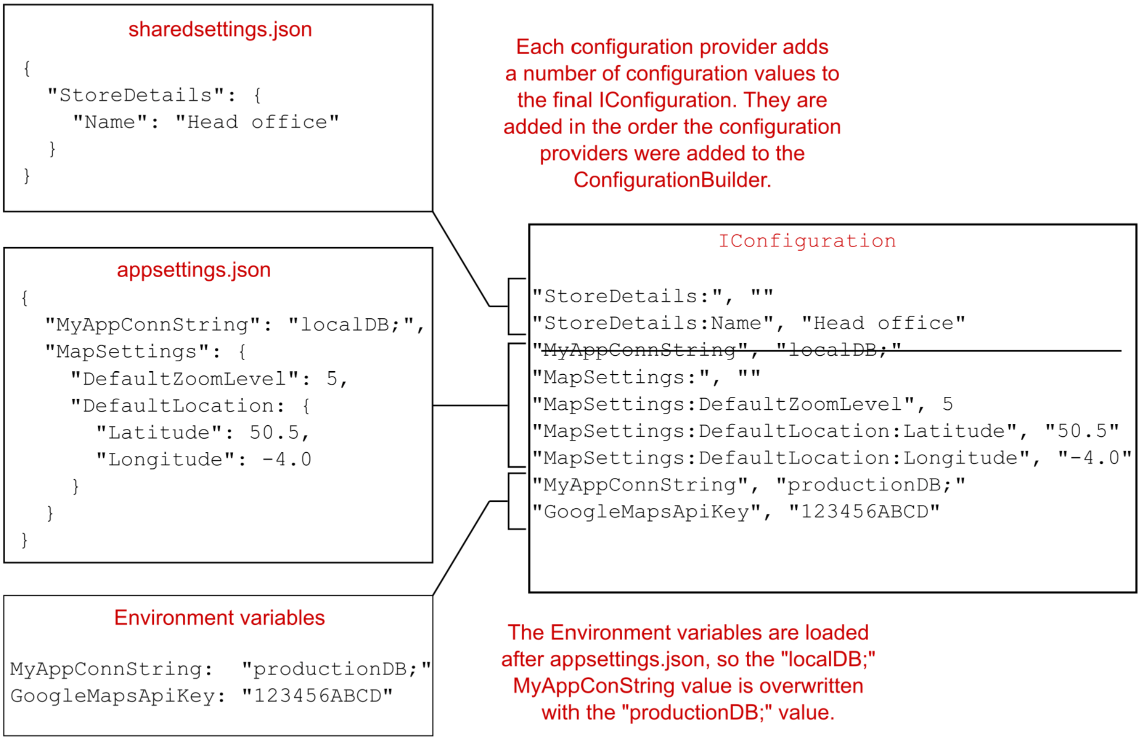
Update your code to load configuration from three different configuration providers—two JSON providers and an environment variable provider—by adding them to `ConfigurationBuilder`. I've only shown the `AddAppConfiguration` method in this listing for brevity.

Listing 11.6 Loading from multiple providers in Startup.cs

```
public class Program
{
    /* Additional Program configuration*/
    public static void AddAppConfiguration(
        HostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        config.Sources.Clear();
        config
           .AddJsonFile("sharedSettings.json", optional: true)   #A
           .AddJsonFile("appsettings.json", optional: true)
           .AddEnvironmentVariables();                           #B
    }
}
```

#A Loads configuration from a different JSON configuration file before the appsettings.json file
#B Adds the machine's environment variables as a configuration provider

This layered design can be useful for a number of things. Fundamentally, it allows you to aggregate configuration values from a number of different sources into a single, cohesive object. To cement this in place, consider the configuration values in figure 11.4.



```
{
  "StoreDetails": {
    "Name": "Head office"
  }
}
```

```
{
  "MyAppConnString": "localDB;",
  "MapSettings": {
    "DefaultZoomLevel": 5,
    "DefaultLocation: {
      "Latitude": 50.5,
      "Longitude": -4.0
    }
  }
}
```

```
MyAppConnString:  "productionDB;"
GoogleMapsApiKey: "123456ABCD"
```

```
"StoreDetails:", ""
"StoreDetails:Name", "Head office"
~~"MyAppConnString", "localDB;"~~
"MapSettings:", ""
"MapSettings:DefaultZoomLevel", 5
"MapSettings:DefaultLocation:Latitude", "50.5"
"MapSettings:DefaultLocation:Longitude", "-4.0"
"MyAppConnString", "productionDB;"
"GoogleMapsApiKey", "123456ABCD"
```

Each configuration provider adds a number of configuration values to the final IConfiguration. They are added in the order the configuration providers were added to the ConfigurationBuilder.

The Environment variables are loaded after appsettings.json, so the "localDB;" MyAppConString value is overwritten with the "productionDB;" value.

Figure 11.4 The final `IConfiguration` includes the values from each of the providers. Both appsettings.json and the environment variables include the `MyAppConnString` key. As the environment variables are added later, that configuration value is used.

Most of the settings in each provider are unique and added to the final `IConfiguration`. But the `"MyAppConnString"` key appears both in appsettings.json and as an environment variable. Because the environment variable provider is added *after* the JSON providers, the environment variable configuration value is used in `IConfiguration`.

The ability to collate configuration from multiple providers is a handy trait on its own, but this design is especially useful when it comes to handling sensitive configuration values, such as connection strings and passwords. The next section shows how to deal with this problem, both locally on your development machine and on production servers.

11.3.3 Storing configuration secrets safely

As soon as you build a nontrivial app, you'll find you have a need to store some sort of sensitive data as a setting somewhere. This could be a password, connection string, or an API key for a remote service, for example.

Storing these values in appsettings.json is generally a bad idea, as you should never commit secrets to source control; the number of secret API keys people have committed to GitHub is scary! Instead, it's much better to store these values outside of your project folder, where they won't get accidentally committed.

You can do this in a few ways, but the easiest and most commonly used approaches are to use environment variables for secrets on your production server and User Secrets locally.

Neither approach is truly secure, in that they don't store values in an encrypted format. If your machine is compromised, attackers will be able to read the stored values as they're stored in plaintext. They're intended to help you avoid committing secrets to source control.

> **TIP** **Azure Key Vault[50] is a secure alternative, in that it stores the values encrypted in Azure. But you will still need to use the following approach for storing the Azure Key Value connection details!**

Whichever approach you choose to store your application secrets, make sure you aren't storing them in source control, if possible. Even private repositories may not stay private forever, so it's best to err on the side of caution!

STORING SECRETS IN ENVIRONMENT VARIABLES IN PRODUCTION

You can add the environment variable configuration provider using the `AddEnvironmentVariables` extension method, as you've already seen in listing 11.6. This adds all of the environment variables on your machine as key-value pairs in the configuration object.

> **REMINDER** **The environment variable provider is added by default in `CreateDefaultBuilder` as you saw in section 11.2.**

You can create the same hierarchical sections in environment variables that you typically see in JSON files by using a colon, `:`, or a double underscore, `__`, to demarcate a section, for example: `MapSettings:MaxNumberOfPoints` or `MapSettings__MaxNumberOfPoints`.

> **TIP** **Some environments, such as Linux, don't allow ":" in environment variables. You must use the double underscore approach in these environments instead. You should always use ":" when *retrieving* values from an `IConfiguration` in your app.**

The environment variable approach is particularly useful when you're publishing your app to a self-contained environment, such as a dedicated server, Azure, or a Docker container. You can set environment variables on your production machine, or on your Docker container, and the

[50] The Azure Key Vault configuration provider is available as a Microsoft.Extensions.Configuration.AzureKeyVault NuGet package. For details on using it in your application, see the documentation at https://docs.microsoft.com/en-us/aspnet/core/security/key-vault-configuration.

provider will read them at runtime, overriding the defaults specified in your appsettings.json files.[51]

For a development machine, environment variables are less useful, as all your apps would be using the same values. For example, if you set the `ConnectionStrings__DefaultConnection` environment variable, then that would be added for *every* app you run locally. That sounds like more of a hassle than a benefit!

For development scenarios, you can use the User Secrets Manager. This effectively adds per-app environment variables, so you can have different settings for each app, but store them in a different location from the app itself.

STORING SECRETS WITH THE USER SECRETS MANAGER IN DEVELOPMENT

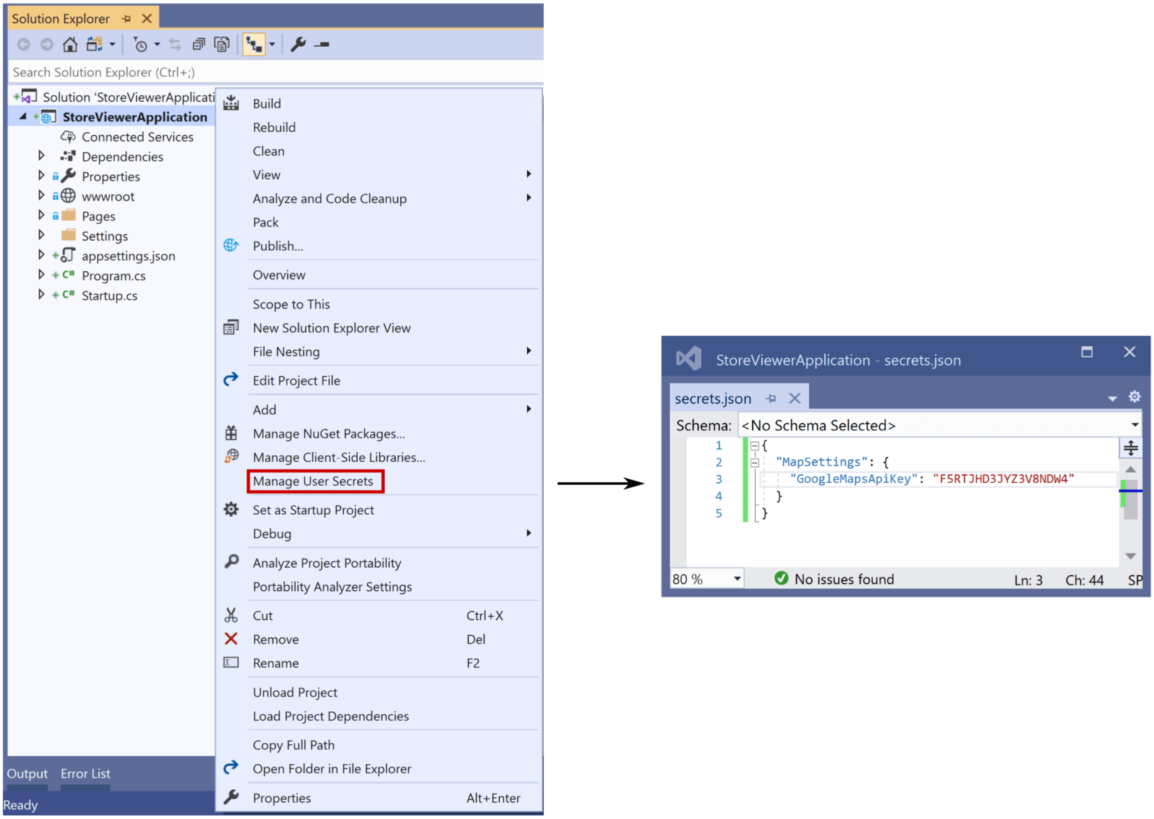
The idea behind User Secrets is to simplify storing per-app secrets outside of your app's project tree. This is similar to environment variables, but you use a unique key per-app to keep the secrets segregated.

> **WARNING** The secrets aren't encrypted, so shouldn't be considered secure. Nevertheless, it's an improvement on storing them in your project folder.

Setting up User Secrets takes a bit more effort than using environment variables, as you need to configure a tool to read and write them, add the User Secrets configuration provider, and define a unique key for your application:

1. ASP.NET Core includes the User Secrets provider by default. The .NET Core SDK also includes a global tool for working with secrets from the command line.
2. If you're using Visual Studio, right-click your project and choose Manage User Secrets. This opens an editor for a secrets.json file in which you can store your key-value pairs, as if it were an appsettings.json file, as shown in figure 11.5.

[51] For instructions on how to set environment variables for your operating system, see the documentation at https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments.



```
{
  "MapSettings": {
    "GoogleMapsApiKey": "F5RTJHD3JYZ3V8NDW4"
  }
}
```

Figure 11.5 Click Manage User Secrets to open an editor for the User Secrets app. You can use this file to store secrets when developing your app locally. These are stored outside your project folder, so won't be committed to source control accidentally.

3. Add a unique identifier to your csproj file. Visual Studio does this automatically when you click Manage User Secrets, but if you're using the command line, you'll need to add it yourself. Typically, you'd use a unique ID, like a GUID:

```
<PropertyGroup>
  <UserSecretsId>96eb2a39-1ef9-4d8e-8b20-8e8bd14038aa</UserSecretsId>
</PropertyGroup>
```

4. If you aren't using Visual Studio, you can add User Secrets using the command line:

```
dotnet user-secrets set "MapSettings:GoogleMapsApiKey" F5RJT9GFHKR7
```

or you can edit the secret.json file directly using your favorite editor. The exact location of this file depends on your operating system and may vary. Check the documentation[52] for details.

Phew, that's a lot of setup, and if you're customizing the `HostBuilder`, you're not done yet! You need to update your app to load the User Secrets at runtime using the `AddUserSecrets` extension method in your ConfigureAppConfiguration method:

```
if(env.IsDevelopment())
{
    configBuilder.AddUserSecrets<Startup>();
}
```

NOTE **You should only use the User Secrets provider in development, *not* in production, so in the snippet above you conditionally add the provider to `ConfigurationBuilder`. In production, you should use environment variables or Azure Key Vault, as discussed earlier. This is all configured correctly by default if you use `Host.CreateDefaultBuilder()`.**

This method has a number of overloads, but the simplest is a generic method that you can call passing your application's `Startup` class as a generic argument. The User Secrets provider needs to read the `UserSecretsId` property that you (or Visual Studio) added to the csproj file. The `Startup` class acts as a simple marker to indicate which assembly contains this property.

NOTE **If you're interested, the User Secrets package uses the `UserSecretsId` property in your csproj file to generate an assembly-level `UserSecretsIdAttribute`. The provider then reads this attribute at runtime to determine the `UserSecretsId` of the app, and hence generates the path to the secrets.json file.**

And there you have it, safe storage of your secrets outside your project folder during development. This might seem like overkill, but if you have anything that you consider remotely sensitive that you need to load into configuration, then I strongly urge you to use environment variables or User Secrets.

It's almost time to leave configuration providers behind, but before we do, I'd like to show you the ASP.NET Core configuration system's party trick: reloading files on the fly.

11.3.4 Reloading configuration values when they change

Besides security, not having to recompile your application every time you want to tweak a value is one of the advantages of using configuration and settings. In the previous version of ASP.NET, changing a setting by editing web.config would cause your app to have to restart.

[52] The Secret Manager Tool .stores the secrets.json file in the user profile. You can read more about the this tool specifically here https://docs.microsoft.com/en-gb/aspnet/core/security/app-secrets and about .NET core tools in general here: https://docs.microsoft.com/en-us/dotnet/core/tools/global-tools.

This beat having to recompile, but waiting for the app to start up before it could serve requests was a bit of a drag.

In ASP.NET Core, you finally get the ability to edit a file and have the configuration of your application automatically update, without having to recompile or restart.

An often cited scenario where you might find this useful is when you're trying to debug an app you have in production. You typically configure logging to one of a number of levels, for example:

- Error
- Warning
- Information
- Debug

Each of these settings is more verbose than the last, but also provides more context. By default, you might configure your app to only log warning and error-level logs in production, so you don't generate too many superfluous log entries. Conversely, if you're trying to debug a problem, you want as much information as possible, so you might want to use the debug log level.

Being able to change configuration at runtime means you can easily switch on extra logs when you encounter an issue and switch them back afterwards by editing your appsettings.json file.

> **NOTE** Reloading is generally only available for file-based configuration providers, as opposed to the environment variable or User Secrets provider.

You can enable the reloading of configuration files when you add any of the file-based providers to your `ConfigurationBuilder`. The `Add*File` extension methods include an overload with a `reloadOnChange` parameter. If this is set to true, the app will monitor the filesystem for changes to the file and will trigger a complete rebuild of the `IConfiguration`, if needs be. This listing shows how to add configuration reloading to the appsettings.json file loaded inside the `AddAppConfiguration` method.

Listing 11.7 Reloading appsettings.json when the file changes

```
public class Program
{
    /* Additional Program configuration*/

    public static void AddAppConfiguration(
        HostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        config.AddJsonFile(
            "appsettings.json",
            optional: true
            reloadOnChange: true);        #A
    }
}
```

#A IConfiguration will be rebuilt if the appsettings.json file changes.

With that in place, any changes you make to the file will be mirrored in the IConfiguration. But as I said at the start of this chapter, IConfiguration isn't the preferred way to pass settings around in your application. Instead, as you'll see in the next section, you should favor strongly typed POCO objects.

11.4 Using strongly typed settings with the options pattern

In this section you'll learn about strongly typed configuration and the Options pattern. This is the preferred way of accessing configuration in ASP.NET Core. By using strongly typed configuration, you can avoid issues with typos when accessing configuration. It also makes classes easier to test, as you can use simple POCO objects for configuration, instead of relying on the IConfiguration abstraction.

Most of the examples I've shown so far have been about how to get values *into* IConfiguration, as opposed to how to *use* them. You've seen that you can access a key using the Configuration["key"] dictionary syntax, but using string keys like this feels messy and prone to typos.

Instead, ASP.NET Core promotes the use of strongly typed settings. These are POCO objects that you define and create and represent a small collection of settings, scoped to a single feature in your app.

The following listing shows both the settings for your store locator component and display settings to customize the homepage of the app. They're separated into two different objects with "MapSettings" and "AppDisplaySettings" keys corresponding to the different areas of the app they impact.

Listing 11.8 Separating settings into different objects in appsettings.json

```
{
  "MapSettings": {              #A
    "DefaultZoomLevel": 6,     #A
    "DefaultLocation": {       #A
      "latitude": 50.500,      #A
      "longitude": -4.000      #A
    }
  },
  "AppDisplaySettings": {            #B
    "Title":  "Acme Store Locator",  #B
    "ShowCopyright": true            #B
  }
}
```

#A Settings related to the store locator section of the app
#B General settings related to displaying the app

The simplest approach to making the homepage settings available in the Index.cshtml Razor Page would be to inject `IConfiguration` into the `PageModel` and access the values using the dictionary syntax; for example:

```
public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        var title = config["HomePageSettings:Title"];
        var showCopyright = bool.Parse(
            config["HomePageSettings:ShowCopyright"]);
    }
}
```

But you don’t want to do this; too many strings for my liking! And that `bool.Parse`? Yuk! Instead, you can use custom strongly typed objects, with all the type safety and IntelliSense goodness that brings.

Listing 11. 9 Injecting strongly typed options into a `PageModel` using `IOptions<T>`

```
public class IndexModel: PageModel
{
    public IndexModel(IOptions<AppDisplaySettings> options)    #A
    {
        AppDisplaySettings settings = options.Value;              #B
        var title = settings.Title;                               #C
        var showCopyright = settings.ShowCopyright;               #D
    }
}
```

#A You can inject a strongly typed options class using the IOptions<> wrapper interface.
#B The Value property exposes the POCO settings object.
#C The settings object contains properties that are bound to configuration values at runtime.
#D The binder can also convert string values directly to primitive types.

The ASP.NET Core configuration system includes a *binder*, which can take a collection of configuration values and *bind* them to a strongly typed object, called an *options class*. This is similar to the concept of model binding from chapter 6, where request values were bound to your POCO binding model classes.

This section shows how to set up the binding of configuration values to a POCO options class and how to make sure it reloads when the underlying configuration values change. We’ll also have a look at the different sorts of objects you can bind.

11.4.1 Introducing the IOptions interface

ASP.NET Core introduced strongly typed settings as a way of letting configuration code adhere to the single responsibility principle and to allow injecting configuration classes as explicit dependencies. They also make testing easier; instead of having to create an instance of `IConfiguration` to test a service, you can create an instance of the POCO options class.

For example, the AppDisplaySettings class shown in the previous example could be simple, exposing just the values related to the homepage:

```
public class AppDisplaySettings
{
    public string Title { get; set; }
    public bool ShowCopyright { get; set; }
}
```

Your options classes need to be non-abstract and have a public parameterless constructor to be eligible for binding. The binder will set any public properties that match configuration values, as you'll see shortly.

> **TIP** You don't just have to primitive types like string and bool, you can use nested complex types too. The options system will bind sections to complex properties. See the associated source code for examples.

To help facilitate the binding of configuration values to your custom POCO options classes, ASP.NET Core introduces the IOptions<T> interface. This is a simple interface with a single property, Value, which contains your configured POCO options class at runtime. Options classes are set up in the ConfigureServices section of Startup, as shown here.

Listing 11. 10 Configuring the options classes using Configure<T> in Startup.cs

```
public IConfiguration Configuration { get; }
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MapSettings>(          #A
        Configuration.GetSection("MapSettings"));     #A

    services.Configure<AppDisplaySettings>(                #B
        Configuration.GetSection("AppDisplaySettings"));   #B
}
```

#A Binds the MapSettings section to the POCO options class MapSettings
#B Binds the AppDisplaySettings section to the POCO options class AppDisplaySettings

> **TIP** You don't have to use the same name for both the section and class as I do in listing 11.10, it's just a convention I like to follow. With this convention you can use the nameof() operator to further reduce the chance of typos, by calling GetSection(nameof(MappSettings)) for example.

Each call to Configure<T> sets up the following series of actions internally:

- Creates an instance of ConfigureOptions<T>, which indicates that IOptions<T> should be configured based on configuration.

  If Configure<T> is called multiple times, then multiple ConfigureOptions<T> objects will be used, all of which can be applied to create the final object, in much the same way as the IConfiguration is built from multiple layers.

- Each ConfigureOptions<T> instance binds a section of IConfiguration to an instance of the T POCO class. This sets any public properties on the options class based on the keys in the provided ConfigurationSection.

  Remember that the section name ("MapSettings" in listing 11.10) can have any value; it doesn't have to match the name of your options class.
- Registers the IOptions<T> interface in the DI container as a singleton, with the final bound POCO object in the Value property.

This last step lets you inject your options classes into controllers and services by injecting IOptions<T>, as you've seen. This gives you encapsulated, strongly typed access to your configuration values. No more magic strings, woo-hoo!

> **WARNING** **If you forget to call Configure<T> and inject IOptions<T> into your services, you won't see any errors, but the T options class won't be bound to anything and will only have default values in its properties.**

The binding of the T options class to ConfigurationSection happens when you first request IOptions<T>. The object is registered in the DI container as a singleton, so it's only bound once.

There's one catch with this setup: you can't use the reloadOnChange parameter I described in section 11.3.4 to reload your strongly typed options classes when using IOptions<T>. IConfiguration will still be reloaded if you edit your appsettings.json files, but it won't propagate to your options class.

If that seems like a step backwards, or even a deal-breaker, then don't worry. IOptions<T> has a cousin, IOptionsSnapshot<T>, for such an occasion.

11.4.2 Reloading strongly typed options with IOptionsSnapshot

In the previous section, you used IOptions<T> to provide strongly typed access to configuration. This provided a nice encapsulation of the settings for a particular service, but with a specific drawback: the options class never changes, even if you modify the underlying configuration file from which it was loaded, for example appsettings.json.

This is often not a problem (you shouldn't really be modifying files on live production servers anyway!), but if you need this functionality, you can use the IOptionsSnapshot<T> interface.

Conceptually, IOptionsSnaphot<T> is identical to IOptions<T> in that it's a strongly typed representation of a section of configuration. The difference is when, and how often, the POCO options objects are created when each of these are used.

- IOptions<T>—The instance is created once, when first needed. It always contains the configuration when the object instance was first created.
- IOptionsSnapshot<T>—A new instance is created when needed, if the underlying configuration has changed since the last instance was created.

IOptionsSnaphot<T> is automatically set up for your options classes at the same time as IOptions<T>, so you can use it in your services in exactly the same way! This listing shows how you could update your IndexModel home page so that you always get the latest configuration values in your strongly typed AppDisplaySettings options class.

Listing 11.11 Injecting reloadable options using IOptionsSnapshot<T>

```
public class IndexModel: PageModel
{
    public IndexModel(
        IOptionsSnapshot<AppDisplaySettings> options)    #A
    {                                                 #B
        AppDisplaySettings settings = options.Value;    #C
        var title = settings.Title;
    }
}
```

#A IOptionsSnapshot<T> will update if the underlying configuration values change.
#B The Value property exposes the POCO settings object, the same as for IOptions<T>.
#C The settings object will match the configuration values at some point, instead of at first run.

Whenever you edit the settings file and cause IConfiguration to be reloaded, IOptionsSnapshot<AppDisplaySettings> will be rebuilt. A new AppDisplaySettings object is created with the new configuration values and will be used for all future dependency injection. Until you edit the file again, of course! It's as simple as that; update your code to use IOptionsSnapshot<T> instead of IOptions<T> wherever you need it.

The final thing I want to touch on in this section is the design of your POCO options classes themselves. These are typically simple collections of properties, but there are a few things to bear in mind so that you don't get stuck debugging why the binding seemingly hasn't worked!

11.4.3 Designing your options classes for automatic binding

I've touched on some of the requirements for your POCO classes for the IOptions<T> binder to be able to populate them, but there are a few rules to bear in mind.

The first key point is that the binder will be creating instances of your options classes using reflection, so your POCO options classes need to:

- Be non-abstract
- Have a default (public parameterless) constructor

If your classes satisfy these two points, the binder will loop through all the properties on your class and bind any it can. In the broadest sense, the binder can bind any property which

- Is public
- Has a getter—the binder won't write set-only properties
- Has a setter or, for complex types, a non-null value
- Is not an indexer

This listing shows an extensive options class, with a whole host of different types of properties, some of which are valid to bind, and some of which aren't.

Listing 11.12 An options class containing binding and nonbinding properties

```
public class TestOptions
{
    public string String { get; set; }                        #A
    public int Integer { get; set; }                          #A
    public SubClass Object { get; set; }                      #A
    public SubClass ReadOnly { get; } = new SubClass();       #A
    public Dictionary<string, SubClass> Dictionary { get; set; }       #B
    public List<SubClass> List { get; set; }                           #B
    public IDictionary<string, SubClass> IDictionary { get; set; }     #B
    public IEnumerable<SubClass> IEnumerable { get; set; }             #B
    public ICollection<SubClass> IEnumerable { get; }                  #B
        = new List<SubClass>();                                        #B

    internal string NotPublic { get; set; }                             #C
    public SubClass SetOnly { set => _setOnly = value; }                #C
    public SubClass NullReadOnly { get; } = null;                       #C
    public SubClass NullPrivateSetter { get; private set; } = null;     #C
    public SubClass this[int i] {                                       #C
        get => _indexerList[i];                                         #C
        set => _indexerList[i] = value;                                 #C
    }
    public List<SubClass> NullList { get; }                             #D
    public Dictionary<int, SubClass> IntegerKeys { get; set; }          #D
    public IEnumerable<SubClass> ReadOnlyEnumerable { get; }            #D
        = new List<SubClass>();                                         #D
    public SubClass _setOnly = null;                    #E
    private readonly List<SubClass> _indexerList        #E
        = new List<SubClass>();                         #E
    public class SubClass
    {
        public string Value { get; set; }
    }
}
```

#A The binder can bind simple and complex object types, and read-only properties with a default.
#B The binder will also bind collections, including interfaces; dictionaries must have string keys.
#C The binder can't bind nonpublic, set-only, null-read-only, or indexer properties.
#D These collection properties can't be bound.
#E The backing fields for SetOnly and Indexer properties

As shown in the listing, the binder generally supports collections—both implementations and interfaces. If the collection property is already initialized, it will use that, but the binder can also create backing fields for them. If your property implements any of the following classes, the binder will create a `List<>` of the appropriate type as the backing object:

- `IReadOnlyList<>`
- `IReadOnlyCollection<>`
- `ICollection<>`

- IEnumerable<>

WARNING **You can't bind to an** IEnumerable<> **property that has already been initialized, as the underlying type doesn't expose an** Add **function! You *can* bind to an** IEnumerable<> **if you leave its initial value as** null**.**

Similarly, the binder will create a Dictionary<,> as the backing field for properties with dictionary interfaces, as long as they use string keys:

- IDictionary<string,>
- IReadOnlyDictionary<string,>

WARNING **You can't bind dictionaries with non-string values, such as** int**. For examples of binding collection types, see the associated source code for this book.**

Clearly there are quite a few nuances here, but if you stick to the simple cases from the preceding example, you'll be fine. Be sure to check for typos in your JSON files!

TIP **The options pattern is most commonly used to bind POCO classes to configuration, but you can also configure your strongly typed settings classes in code, by providing a lambda to the** Configure **function, for example** services.Configure<TestOptions>(opt => opt.Value=true).

That brings us to the end of this section on strongly typed settings. In the next section, we'll look at how you can dynamically change your settings at runtime, based on the environment in which your app is running.

11.5 Configuring an application for multiple environments

In this section you'll learn about hosting environments in ASP.NET Core. You'll learn how to set and determine which environment an application is running in, and how to change which configuration values are used based on the environment. This lets you easily switch between different sets of configuration values in production compared to development, for example.

Any application that makes it to production will likely have to run in multiple environments. For example, if you're building an application with database access, you'll probably have a small database running on your machine that you use for development. In production, you'll have a completely different database running on a server somewhere else.

Another common requirement is to have different amounts of logging depending on where your app is running. In development, it's great to generate lots of logs as it helps debugging, but once you get to production, too much logging can be overwhelming. You'll want to log warnings and errors, maybe information-level logs, but definitely not debug-level logs!

To handle these requirements, you need to make sure your app loads different configuration values depending on the environment it's running in; load the production database connection string when in production and so on. You need to consider three aspects:

- How does your app identify which environment it’s running in?
- How do you load different configuration values based on the current environment?
- How can you change the environment for a particular machine?

This section tackles each of these questions in turn, so you can easily tell your development machine apart from your production servers and act accordingly.

11.5.1 Identifying the hosting environment

As you saw in section 11.2, the `ConfigureHostingConfiguration` method on `HostBuilder` is where you define how your application calculates the hosting environment. By default, `CreateDefaultBuilder` uses, perhaps unsurprisingly, an environment variable to identify the current environment! The `HostBuilder` looks for a magic environment variable called `ASPNETCORE_ENVIRONMENT` and uses it to create an `IHostEnvironment` object.

> **NOTE** **You can use either the `DOTNET_ENVIRONMENT` or `ASPNETCORE_ENVIRONMENT` environment variables. The `ASPNETCORE_` value overrides the `DOTNET_` value if both are set. I use the `ASPNETCORE_` version throughout this book.**

The `IHostEnvironment` interface exposes a number of useful properties about the running context of your app. Some of these you’ve already seen, such as `ContentRootPath`, which points to the folder containing your application’s content files; for example, the appsettings.json files. The property you’re interested in here is `EnvironmentName`.

The `IHostEnvironment.EnvironmentName` property is set to the value of the `ASPNETCORE_ENVIRONMENT` environment variable, so it can be anything, but you should stick to three commonly used values in most cases:

- `"Development"`
- `"Staging"`
- `"Production"`

ASP.NET Core includes several helper methods for working with these three values, so you’ll have an easier time if you stick to them. In particular, whenever you’re testing whether your app is running in a particular environment, you should use one of the following extension methods:

- `IHostEnvironment.IsDevelopment()`
- `IHostEnvironment.IsStaging()`
- `IHostEnvironment.IsProduction()`
- `IHostEnvironment.IsEnvironment(string environmentName)`

These methods all make sure they do case-insensitive checks of the environment variable, so you don’t get any wonky errors at runtime if you don’t capitalize the environment variable value.

TIP Where possible, use the `IHostEnvironment` extension methods over direct string comparison with `EnvironmentValue`, as they provide case-insensitive matching.

`IHostEnvironment` doesn't do anything other than expose the details of your current environment, but you can use it in a number of different ways. In chapter 8, you saw the Environment Tag Helper, which you used to show and hide HTML based on the current environment. Now you know where it was getting its information—`IHostEnvironment`.

You can use a similar approach to customize which configuration values you load at runtime by loading different files when running in development versus production. This is common and is included out of the box in most ASP.NET Core templates, as well as in the `CreateDefaultBuilder` helper method.

11.5.2 Loading environment-specific configuration files

The `EnvironmentName` value is determined early in the process of bootstrapping your application, before the `ConfigurationBuilder` passed to `ConfigureAppConfiguration` is created. This means you can dynamically change which configuration providers are added to the builder, and hence which configuration values are loaded when the `IConfiguration` is built.

A common pattern is to have an optional, environment-specific appsettings.ENVIRONMENT.json file that's loaded after the default appsettings.json file. This listing shows how you could achieve this if you're customizing the `ConfigureAppConfiguration` method in Program.cs.

Listing 11.13 Adding environment-specific `appsettings.json` files

```
public class Program
{
    public static void AddAppConfiguration(
        HostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        var env = hostingContext.HostingEnvironment;     #A
        config

            .AddJsonFile(
                "appsettings.json",
                optional: false)                          #B
            .AddJsonFile                                 #C
                $"appsettings.{env.EnvironmentName}.json",   #C
                optional: true);                              #C
    }
}
```

#A The current IHostEnvironment is available on HostBuilderContext.
#B It's common to make the base appsettings.json compulsory.
#C Adds an optional environment-specific JSON file where the filename varies with the environment

With this pattern, a global appsettings.json file contains settings applicable to most environments. Additional, optional, JSON files called appsettings.Development.json, appsettings.Staging.json, and appsettings.Production.json are subsequently added to ConfigurationBuilder, depending on the current EnvironmentName.

Any settings in these files will overwrite values from the global appsettings.json, if they have the same key, as you've seen previously. This lets you do things like set the logging to be verbose in the development environment only and switch to more selective logs in production.

Another common pattern is to completely add or remove configuration providers depending on the environment. For example, you might use the User Secrets provider when developing locally, but Azure Key Vault in production. This listing shows how you can use IHostEnvironment to conditionally include the User Secrets provider in development only.

Listing 11.14 Conditionally including the User Secrets configuration provider

```
public class Program
{
    /* Additional Program configuration*/
    public static void AddAppConfiguration(
        HostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        var env = hostingContext.HostingEnvironment
        config
            .AddJsonFile(
                "appsettings.json",
                optional: false)
            .AddJsonFile(
                $"appsettings.{env.EnvironmentName}.json",
                optional: true);
         if(env.IsDevelopment())                   #A
         {
             builder.AddUserSecrets<Startup>();    #B
        }
    }
}
```

#A Extension methods make checking the environment simple and explicit.
#B In Staging and Production, the User Secrets provider wouldn't be used.

Another common place to customize your application based on the environment is when setting up your middleware pipeline. In chapter 3, you learned about DeveloperExceptionPageMiddleware and how you should use it when developing locally. The following listing shows how you can use IHostEnvironment to control your pipeline in this way, so that when you're in Staging or Production, your app uses ExceptionHandlerMiddleware instead.

Listing 11.15 Using the hosting environment to customize your middleware pipeline

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())                   #A
    {                                          #A
        app.UseDeveloperExceptionPage();       #A
    }                                          #A
    else                                           #B
    {                                              #B
        app.UseExceptionHandler("/Error");         #B
    }                                              #B

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });

}
```

#A In Development, DeveloperExceptionPageMiddleware is added to the pipeline.
#B In Staging or Production, the pipeline uses ExceptionHandlerMiddleware instead.

> **NOTE** In listing 11.15 we injected `IWebHostEnvironment` instead of `IHostEnvironment`. This interface extends `IHostEnvironment` by adding the `WebRootPath` property—the path to the wwwroot folder in your application. You'll learn more about the difference between these interfaces in chapter 20.

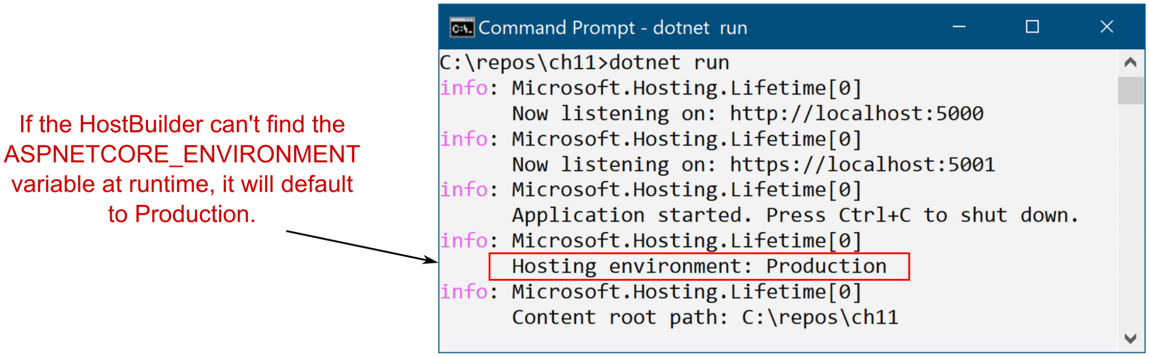
You can inject `IHostEnvironment` anywhere in your app, but I'd advise against using it in your own services, outside of `Startup` and `Program`. It's far better to use the configuration providers to customize strongly typed settings based on the current hosting environment, and inject these settings into your application instead.

As useful as it is, setting `IHostEnvironment` with an environment variable can be a little cumbersome if you want to switch back and forth between different environments during testing. Personally, I'm always forgetting how to set environment variables on the various operating systems I use. The final skill I'd like to teach you is how to set the hosting environment when you're developing locally.

11.5.3 Setting the hosting environment

In this section, I'll show you a couple of ways to set the hosting environment when you're developing. These makes it easy to test a specific app's behavior in different environments, without having to change the environment for all the apps on your machine.

If your ASP.NET Core application can't find an `ASPNETCORE_ENVIRONMENT` environment variable when it starts up, it defaults to a production environment, as shown in figure 11.6. This means that when you deploy to production, you'll be using the correct environment by default.



```
C:\repos\ch11>dotnet run
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\repos\ch11
```

If the HostBuilder can't find the ASPNETCORE_ENVIRONMENT variable at runtime, it will default to Production.

Figure 11.6 By default, ASP.NET Core applications run in the Production hosting environment. You can override this by setting the `ASPNETCORE_ENVIRONMENT` variable.

> **TIP** By default, the current hosting environment is logged to the console at startup, which can be useful for quickly checking that the environment variable has been picked up correctly.

Another option is to use a launchSettings.json file to control the environment. All the default ASP.NET Core applications include this file in the Properties folder. LaunchSettings.json defines "profiles" for running your application.

> **TIP** Profiles can be used to run your application with different environment variables. You can also use profiles to emulate running on Windows behind IIS by using the IIS Express profile. Personally, I rarely use this profile, even on Windows, and always choose the "project" profile.

A typical launchSettings.json file is shown in the following listing, which defines two profiles: "IIS Express", and "StoreViewerApplication". The latter profile is equivalent to using `dotnet run` to run the project, and is conventionally named the same as the project containing the launchSettings.json file.

Listing 11.16 A typical launchSettings.json file defining two profiles

```
{
  "iisSettings": {                                       #A
    "windowsAuthentication": false,                      #A
    "anonymousAuthentication": true,                     #A
    "iisExpress": {                                      #A
      "applicationUrl": "http://localhost:53846",        #A
      "sslPort": 44399                                   #A
    }
  },
  "profiles": {
    "IIS Express": {                                     #B
      "commandName": "IISExpress",                       #B
      "launchBrowser": true,                             #C
      "environmentVariables": {                          #D
```

```
        "ASPNETCORE_ENVIRONMENT": "Development"            #D
      }                                                   #D
    },
    "StoreViewerApplication": {                           #E
      "commandName": "Project",                           #E
      "launchBrowser": true,                         #C
      "environmentVariables": {                      #F
        "ASPNETCORE_ENVIRONMENT": "Development"      #F
      },                                             #F
      "applicationUrl":
          "https://localhost:5001;http://localhost:5000"     #G
    }
  }
}
```

#A Defines settings for when running behind IIS or using the IIS Express profile
#B The IIS Express profile is used by default in Visual Studio on Windows
#C If true, launches the browser when you run the application
#D Defines custom environment variables for the profile. Sets the environment to Development
#E The “project” profile, equivalent to calling dotnet run on the project
#F Each profile can have different environment variables
#G Defines the URLs the application will listen on in this profile

The advantage of using the launchSettings.json file locally is it allows you to set “local” environment variables for a project. For example, in listing 11.16, the environment is set to the Development environment. This lets you use different environment variables for each project, and even for each profile, and store them in source control.

You can choose a profile to use in Visual Studio by selecting from the dropdown list next to the Debug button on the toolbar, as shown in figure 11.7. You can choose a profile to run from the command line using `dotnet run --launch-profile <Profile Name>`. If you don’t specify a profile, the first “Project” type profile is used. If you don’t want to use *any* profile, you must explicitly ignore the launchSettings.json file, by using `dotnet run --no-launch-profile`.

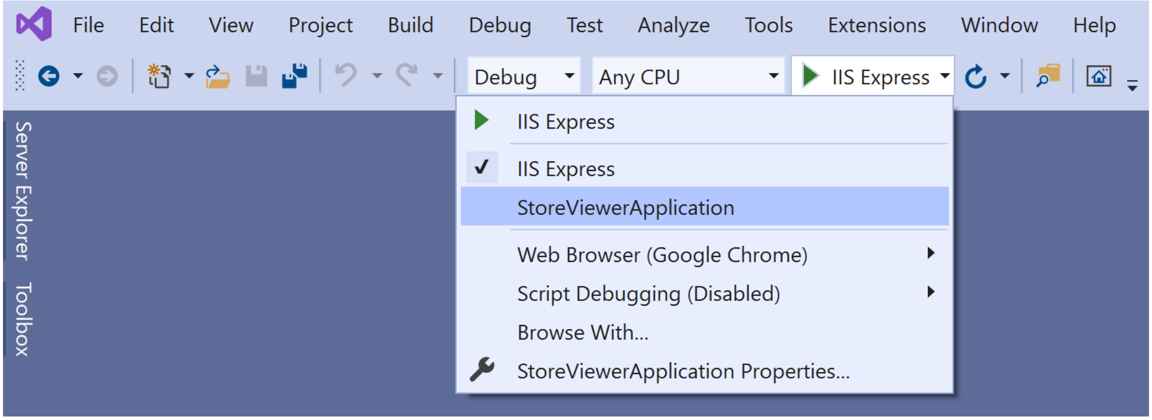
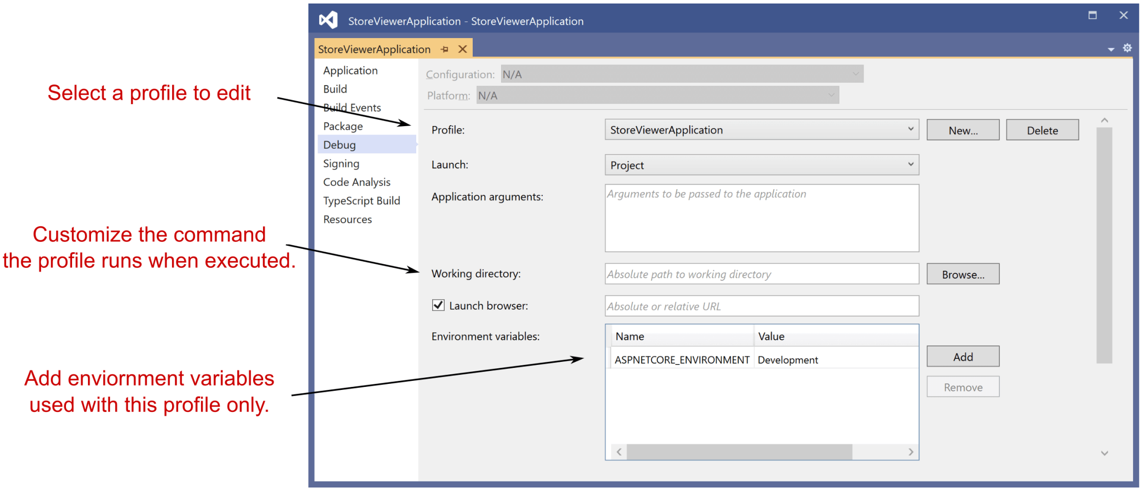


Figure 11.7 You can choose the profile to use from Visual Studio by selecting from the Debug dropdown. Visual Studio defaults to using the IIS Express profile. The default profile running with `dotnet run` is the first "project" profile—StoreViewerApplication in this case.

If you're using Visual Studio, you can also edit the launchSettings.json file visually: double-click the Properties node and choose the Debug tab. You can see in figure 11.8 that the `ASPNETCORE_ENVIRONMENT` is set to Development: any changes made in this tab are mirrored in launchSettings.json.



Select a profile to edit

Customize the command the profile runs when executed.

Add enviornment variables used with this profile only.

Configuration: N/A

Platform: N/A

Profile: StoreViewerApplication

Launch: Project

Application arguments: *Arguments to be passed to the application*

Working directory: *Absolute path to working directory*

☑ Launch browser: *Absolute or relative URL*

Environment variables:

Name	Value
ASPNETCORE_ENVIRONMENT	Development

Figure 11.8 You can use Visual Studio to edit the launchSettings.json file if you prefer. Changes will be mirrored between the launchSettings.json file and the Properties dialog.

The launchSettings.json file is intended for local development only; by default the file isn't deployed to production servers. While you *can* deploy and use the file in production, it's generally not worth the hassle. Environment variables are a better fit.

One final trick I've used to set the environment in production is to use command line arguments. For example, you could set the environment to Staging using the following

```
dotnet run --no-launch-profile --environment Staging
```

Note that you also have to pass `--no-launch-profile` if there's a launchSettings.json file, otherwise the values in the file take precedence.

That brings us to the end of this chapter on configuration. Configuration isn't glamorous, but it's an essential part of all apps. The ASP.NET Core configuration provider model handles a wide range of scenarios, letting you store settings and secrets in a variety of locations.

Simple settings can be stored in appsettings.json, where they're easy to tweak and modify during development, and can be overwritten by using environment-specific JSON files. Meanwhile, your secrets and sensitive settings can be stored outside the project file, in the User Secrets manager, or as environment variables. This gives you both flexibility and safety—as long as you don't go writing your secrets to appsettings.json!

In the next chapter, we'll take a brief look at the new object relational mapper that fits well with ASP.NET Core: Entity Framework Core. We'll only be getting a taste of it in this book, but you'll learn how to load and save data, build a database from your code, and migrate the database as your code evolves.

11.6 Summary

- Anything that could be considered a setting or a secret is normally stored as a configuration value.
- ASP.NET Core uses configuration providers to load key-value pairs from a variety of sources. Applications can use many different configuration providers.
- `ConfigurationBuilder` describes how to construct the final configuration representation for your app and `IConfiguration` holds the configuration values themselves.
- You create a configuration object by adding configuration providers to an instance of `ConfigurationBuilder` using extension methods such as `AddJsonFile()`. `HostBuilder` creates the `ConfigurationBuilder` instance for you and calls `Build()` to create an instance of `IConfiguration`.
- ASP.NET Core includes built-in providers for JSON files, XML files, environment files, and command-line arguments, among others. NuGet packages exist for many other providers, such as YAML files and Azure KeyVault.
- The order in which you add providers to `ConfigurationBuilder` is important; subsequent providers replace the values of settings defined in earlier providers.
- Configuration keys aren't case-sensitive.

- You can retrieve settings from IConfiguration directly using the indexer syntax, for example Configuration["MySettings:Value"].
- The CreateDefaultBuilder method configures JSON, environment variables, command-line arguments, and User Secret providers for you. You can customize the configuration providers used in your app by calling ConfigureAppConfiguration.
- In production, store secrets in environment variables. These can be loaded after your file-based settings in the configuration builder.
- On development machines, the User Secrets Manager is a more convenient tool than using environment variables. It stores secrets in your OS user's profile, outside the project folder.
- Be aware that neither environment variables nor the User Secrets Manager tool encrypt secrets, they merely store them in locations that are less likely to be made public, as they're outside your project folder.
- File-based providers, such as the JSON provider, can automatically reload configuration values when the file changes. This allows you to update configuration values in real time, without having to restart your app.
- Use strongly typed POCO options classes to access configuration in your app.
- Use the Configure<T>() extension method in ConfigureServices to bind your POCO options objects to ConfigurationSection.
- You can inject the IOptions<T> interface into your services using DI. You can access the strongly typed options object on the Value property.
- You can configure IOptions<T> objects in code instead of using configuration values by passing a lambda to the Configure() method.
- If you want to reload your POCO options objects when your configuration changes, use the IOptionsSnapshot<T> interface instead.
- Applications running in different environments, Development versus Production for example, often require different configuration values.
- ASP.NET Core determines the current hosting environment using the ASPNETCORE_ENVIRONMENT environment variable. If this variable isn't set, the environment is assumed to be Production.
- You can set the hosting environment locally by using the launchSettings.json. This allows you to scope environment variables to a specific project.
- The current hosting environment is exposed as an IHostEnvironment interface. You can check for specific environments using IsDevelopment(), IsStaging(), and IsProduction().
- You can use the IHostEnvironment object to load files specific to the current environment, such as appsettings.Production.json.

12
Saving data with Entity Framework Core

This chapter includes

- **What Entity Framework Core is and why you should use it**
- **Adding Entity Framework Core to an ASP.NET Core application**
- **Building a data model and using it to create a database**
- **Querying, creating, and updating data using Entity Framework Core**

Most applications that you'll build with ASP.NET Core will require storing and loading some kind of data. Even the examples so far in this book have assumed you have some sort of data store—storing exchange rates, user shopping carts, or the locations of physical main street stores. I've glossed over this for the most part but, typically, you'll store this data in a database.

Working with databases can often be a rather cumbersome process. You have to manage connections to the database, translate data from your application to a format the database can understand, as well as handle a plethora of other subtle issues.

You can manage this complexity in a variety of ways, but I'm going to focus on using a library built primarily for .NET Core: Entity Framework Core (EF Core). EF Core is a library that lets you quickly and easily build database access code for your ASP.NET Core applications. It's modeled on the popular Entity Framework 6.x library, but has significant changes that mean it stands alone in its own right and is more than an upgrade.

The aim of this chapter is to provide a quick overview of EF Core and how you can use it in your applications to quickly query and save to a database. You'll have enough knowledge to connect your app to a database, and to manage schema changes to the database, but I won't be going into great depth on any topics.

NOTE For an in-depth look at EF Core I recommend, *Entity Framework Core in Action, Second Edition* by Jon P Smith (Manning, 2021). Alternatively, you can read about EF Core and its cousin, Entity Framework, on the Microsoft documentation website at https://docs.microsoft.com/ef/core/.

Section 12.1 introduces EF Core and explains why you might want to use it in your applications. You'll learn how the design of EF Core helps you to quickly iterate on your database structure and reduce the friction of interacting with a database.

In section 12.2, you'll learn how to add EF Core to an ASP.NET Core app and configure it using the ASP.NET Core configuration system. You'll see how to build a model for your app that represents the data you'll store in the database and how to hook it into the ASP.NET Core DI container.

NOTE For this chapter, and the rest of the book, I'm going to be using SQL Server Express' LocalDB feature. This is installed as part of Visual Studio 2019 (when you choose the ASP.NET and Web Development workload) and provides a lightweight SQL Server engine.[53] Very little of the code is specific to SQL Server, so you should be able to follow along with a different database if you prefer. The code sample for the book includes a version using SQLite, for example.

No matter how carefully you design your original data model, the time will come when you need to change it. In section 12.3, I show how you can easily update your model and apply these changes to the database itself, using EF Core for all the heavy lifting.

Once you have EF Core configured and a database created, section 12.4 shows how to use EF Core in your application code. You'll see how to create, read, update, and delete (CRUD) records, as well as some of the patterns to use when designing your data access.

In section 12.5, I highlight a few of the issues you'll want to take into consideration when using EF Core in a production app. A single chapter on EF Core can only offer a brief introduction to all of the related concepts though, so if you choose to use EF Core in your own applications—especially if this is your first time using such a data access library—I strongly recommend reading more once you have the basics from this chapter.

Before we get into any code, let's look at what EF Core is, what problems it solves, and when you might want to use it.

12.1 Introducing Entity Framework Core

Database access code is ubiquitous across web applications. Whether you're building an e-commerce app, a blog, or the Next Big Thing™, chances are you'll need to interact with a database.

[53] You can read more about LocalDB at https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-express-localdb.

Unfortunately, interacting with databases from app code is often a messy affair and there are many different approaches you can take. For example, something as simple as reading data from a database requires handling network connections, writing SQL statements, and handling variable result data. The .NET ecosystem has a whole array of libraries you can use for this, ranging from the low-level ADO.NET libraries, to higher-level abstractions like EF Core.

In this section, I describe what EF Core is and the problem it’s designed to solve. I cover the motivation behind using an abstraction such as EF Core, and how it helps to bridge the gap between your app code and your database. As part of that, I present some of the trade-offs you’ll make by using it in your apps, which should help you decide if it’s right for your purposes. Finally, we’ll take a look at an example EF Core mapping, from app code to database, to get a feel for EF Core’s main concepts.

12.1.1 What is EF Core?

EF Core is a library that provides an object-oriented way to access databases. It acts as an *object-relational mapper* (ORM), communicating with the database for you and mapping database responses to .NET classes and objects, as shown in figure 12.1.

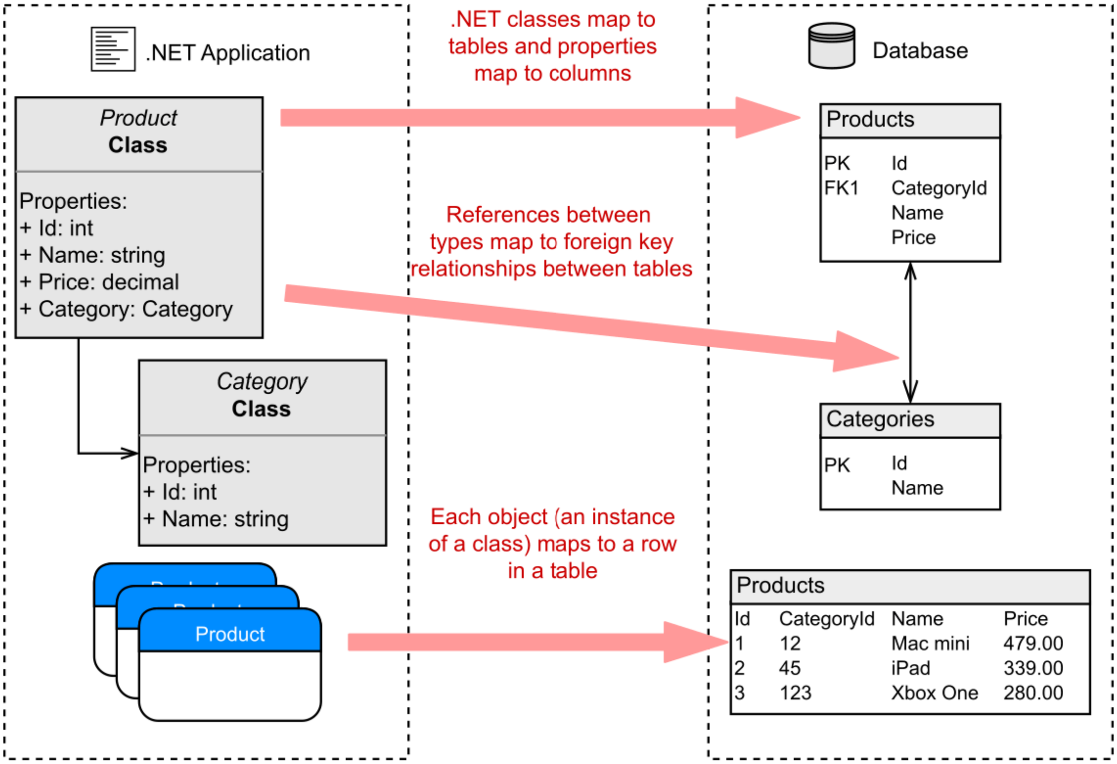


Figure 12.1 EF Core maps .NET classes and objects to database concepts such as tables and rows.

> **DEFINITION** With an *object-relational mapper* (ORM), you can manipulate a database using object-oriented concepts such as classes and objects by mapping them to database concepts such as tables and columns.

EF Core is based on, but is distinct from, the existing Entity Framework libraries (currently, up to version 6.x). It was built as part of the .NET Core push to work cross-platform, but with additional goals in mind. In particular, the EF Core team wanted to make a highly performant library that could be used with a wide range of databases.

There are many different types of databases, but probably the most commonly used family is *relational* databases, accessed using Structured Query Language (SQL). This is the bread and butter of EF Core; it can map Microsoft SQL Server, MySQL, Postgres, and many other relational databases. It even has a cool in-memory feature you can use when testing to create a temporary database. EF Core uses a provider model, so that support for other relational databases can be plugged in later, as they become available.

NOTE As of .NET Core 3.0, EF Core now also works with nonrelational, *NoSQL*, or *document* databases like Cosmos DB too. I'm only going to consider mapping to relational databases in this book however, as that's the most common requirement in my experience. Historically, most data access, especially in the .NET ecosystem, has been using relational databases, so it generally remains the most popular approach.

That covers what EF Core is, but it doesn't dig into why you'd want to use it. Why not access the database directly using the traditional ADO.NET libraries? Most of the arguments for using EF Core can be applied to ORMs in general, so what are the advantages of an ORM?

12.1.2 Why use an object-relational mapper?

One of the biggest advantages an ORM brings is the speed with which you can develop an application. You can stay in the familiar territory of object-oriented .NET, in many cases without ever needing to directly manipulate a database or write custom SQL.

Imagine you have an e-commerce site and you want to load the details of a product from the database. Using low-level database access code, you'd have to open a connection to the database, write the necessary SQL using the correct table and column names, read the data over the connection, create a POCO to hold the data, and manually set the properties on the object, converting the data to the correct format as you go. Sounds painful, right?

An ORM, such as EF Core, takes care of most of this for you. It handles the connection to the database, generating the SQL, and mapping data back to your POCO objects. All you need to provide is a LINQ query describing the data you want to retrieve.

ORMs serve as high-level abstractions over databases, so they can significantly reduce the amount of plumbing code you need to write to interact with a database. At the most basic level, they take care of mapping SQL statements to objects and vice versa, but most ORMs take this a step further and provide additional features.

ORMs like EF Core keep track of which properties have changed on any objects they rehydrate from the database. This lets you load an object from the database by mapping it from a database table, modify it in .NET code, and then ask the ORM to update the associated record in the database. The ORM will work out which properties have changed and issue update statements for the appropriate columns, saving you a bunch of work.

As is so often the case in software development, using an ORM has its drawbacks. One of the biggest advantages of ORMs is also their Achilles heel—they hide you from the database. Sometimes, this high level of abstraction can lead to problematic database query patterns in your apps. A classic example is the *N+1* problem, where what should be a single database request turns into separate requests for every single row in a database table.

Another commonly cited drawback is performance. ORMs are abstractions over a number of concepts, and so inherently do more work than if you were to hand craft every piece of data access in your app. Most ORMs, EF Core included, trade off some degree of performance for ease of development.

That said, if you're aware of the pitfalls of ORMs, you can often drastically simplify the code required to interact with a database. As with anything, if the abstraction works for you, use it, otherwise, don’t. If you only have minimal database access requirements, or you need the best performance you can get, then an ORM such as EF Core may not be the right fit.

An alternative is to get the best of both worlds: use an ORM for the quick development of the bulk of your application, and then fall back to lower level APIs such as ADO.NET for those few areas that prove to be the bottlenecks in your application. That way, you can get good-enough performance with EF Core, trading off performance for development time, and only optimize those areas that need it.

Even if you do decide to use an ORM in your app, there are many different ORMs available for .NET, of which EF Core is one. Whether EF Core is right for you will depend on the features you need and the trade-offs you’re willing to make to get them. The next section compares EF Core to Microsoft’s other offering, Entity Framework, but there many other alternatives you could consider, such as Dapper and NHibernate, each with their own set of trade-offs.

12.1.3 When should you choose EF Core?

Microsoft designed EF Core as a reimagining of the mature Entity Framework 6.x (EF 6.x) ORM, which it released in 2008. With ten years of development behind it, EF 6.x is a stable and feature-rich ORM.

In contrast, EF Core, is a comparatively new project. The APIs of EF Core are designed to be close to that of EF 6.x—though they aren’t identical—but the core components have been completely rewritten. You should consider EF Core as distinct from EF 6.x; upgrading directly from EF 6.x to EF Core is nontrivial.

Microsoft supports both EF Core and EF 6.x, and both will see ongoing improvements, so which should you choose? You need to consider a number of things:

- *Cross platform*—EF Core targets .NET Standard, so it can be used in cross-platform apps that target .NET Core. Since version 6.3, EF 6.x is *also* cross platform, with some limitations when running on .NET Core, such as no designer support.
- *Database providers*—Both EF 6.x and EF Core let you connect to various database types by using pluggable providers. EF Core has a growing number of providers, but there aren’t as many for EF 6.x, especially if you want to run EF 6.x on .NET Core. If there isn’t a provider for the database you’re using, that’s a bit of a deal breaker!
- *Performance*—The performance of EF 6.x has been a bit of a black mark on its record, so EF Core aims to rectify that. EF Core is designed to be fast and lightweight, significantly outperforming EF 6.x. But it’s unlikely to ever reach the performance of a more lightweight ORM, such as Dapper, or handcrafted SQL statements.
- *Features*—Features are where you’ll find the biggest disparity between EF 6.x and EF Core. EF Core has some features that EF 6.x doesn’t have (batching statements, client-side key generation, in-memory database for testing), but EF 6.x is much more feature-rich than EF Core.

  At the time of writing, EF Core has excellent feature parity, with missing features including stored procedure mapping, and many-to-many relationships without a join entity. On top of that, EF Core is under active development, so new features will no doubt appear soon.[54] In contrast, EF 6.x will likely only see incremental improvements and bug fixes, rather than major feature additions.

Whether these trade-offs and limitations are a problem for you will depend a lot on your specific app. It's a lot easier to start a new application bearing these limitations in mind than trying to work around them later.

TIP EF Core isn't recommended for everyone, but it's recommended over EF 6.x for new applications. Be sure you understand the trade-offs involved, and keep an eye on the guidance from the EF team here: https://docs.microsoft.com/en-us/ef/efcore-and-ef6/choosing.

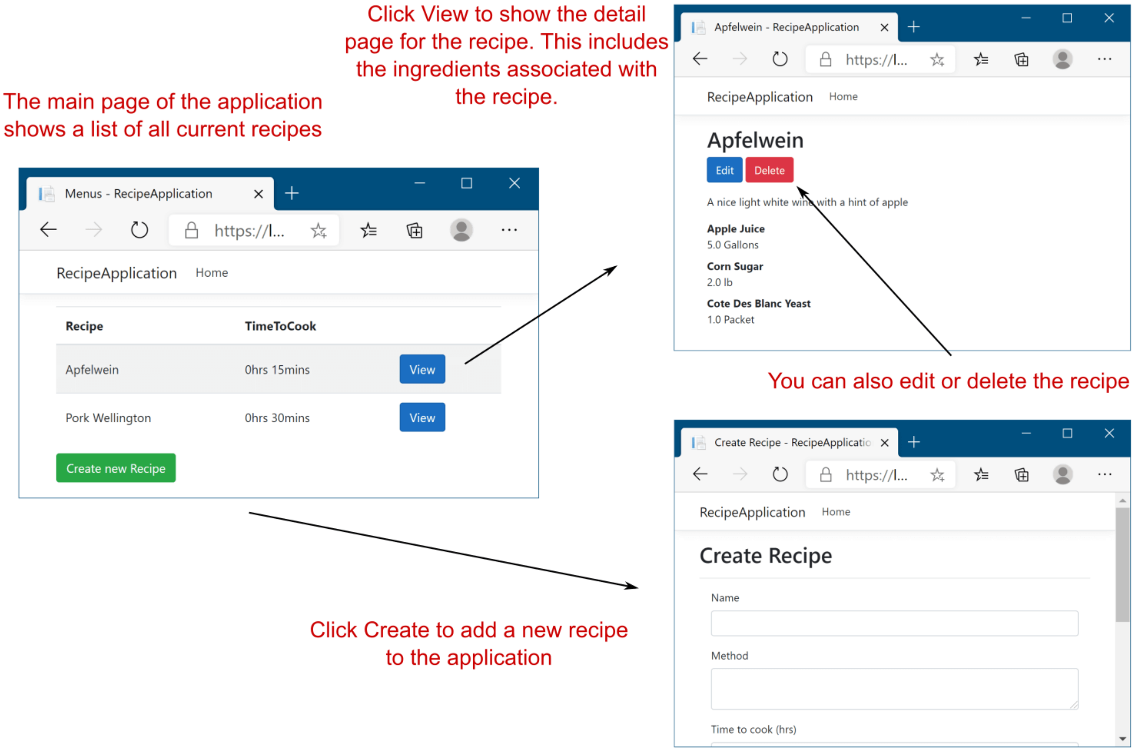
If you're working on a new ASP.NET Core application, you want to use an ORM for rapid development, and you don't require any of the unavailable features, then EF Core is a great contender. It's also supported out of the box by various other subsystems of ASP.NET Core. For instance, in chapter 14 you'll see how to use EF Core with the ASP.NET Core Identity authentication system for managing users in your apps.

Before we get into the nitty-gritty of using EF Core in your app, I'll describe the application we're going to be using as the case study for this chapter. We'll go over the application and database details and how to use EF Core to communicate between the two.

12.1.4 Mapping a database to your application code

EF Core focuses on the communication between an application and a database, so to show it off, we need an application! This chapter uses the example of a simple cooking app that lists recipes, and lets you view a recipe's ingredients, as shown in figure 12.2. Users can browse for recipes, add new ones, edit recipes, and delete old ones.

[54] For a detailed list of feature differences between EF 6.x and EF Core, see the documentation at https://docs.microsoft.com/en-us/ef/efcore-and-ef6/features.



Click View to show the detail page for the recipe. This includes the ingredients associated with the recipe.

The main page of the application shows a list of all current recipes

Recipe	TimeToCook	
Apfelwein	0hrs 15mins	View
Pork Wellington	0hrs 30mins	View

Apfelwein

A nice light white wine with a hint of apple

Apple Juice
5.0 Gallons

Corn Sugar
2.0 lb

Cote Des Blanc Yeast
1.0 Packet

You can also edit or delete the recipe

Create Recipe

Name

Method

Time to cook (hrs)

Click Create to add a new recipe to the application

Figure 12.2 The cookery app lists recipes. You can view, update, and delete recipes, or create new ones.

This is obviously a simple application, but it contains all the database interactions you need with its two *entities*: `Recipe` and `Ingredient`.

> **DEFINITION** An *entity* is a .NET class that's mapped by EF Core to the database. These are classes you define, typically as POCO classes, that can be saved and loaded by mapping to database tables using EF Core.

When you interact with EF Core, you'll be primarily using POCO entities and a *database context* that inherits from the `DbContext` EF Core class. The entity classes are the object-oriented representations of the tables in your database; they represent the data you want to store in the database. You use the `DbContext` in your application to both configure EF Core and to access the database at runtime.

> **NOTE** You can potentially have multiple `DbContext`s in your application and can even configure them to integrate with different databases.

When your application first uses EF Core, EF Core creates an internal representation of the database, based on the DbSet<T> properties on your application's DbContext and the entity classes themselves, as shown in figure 12.3.

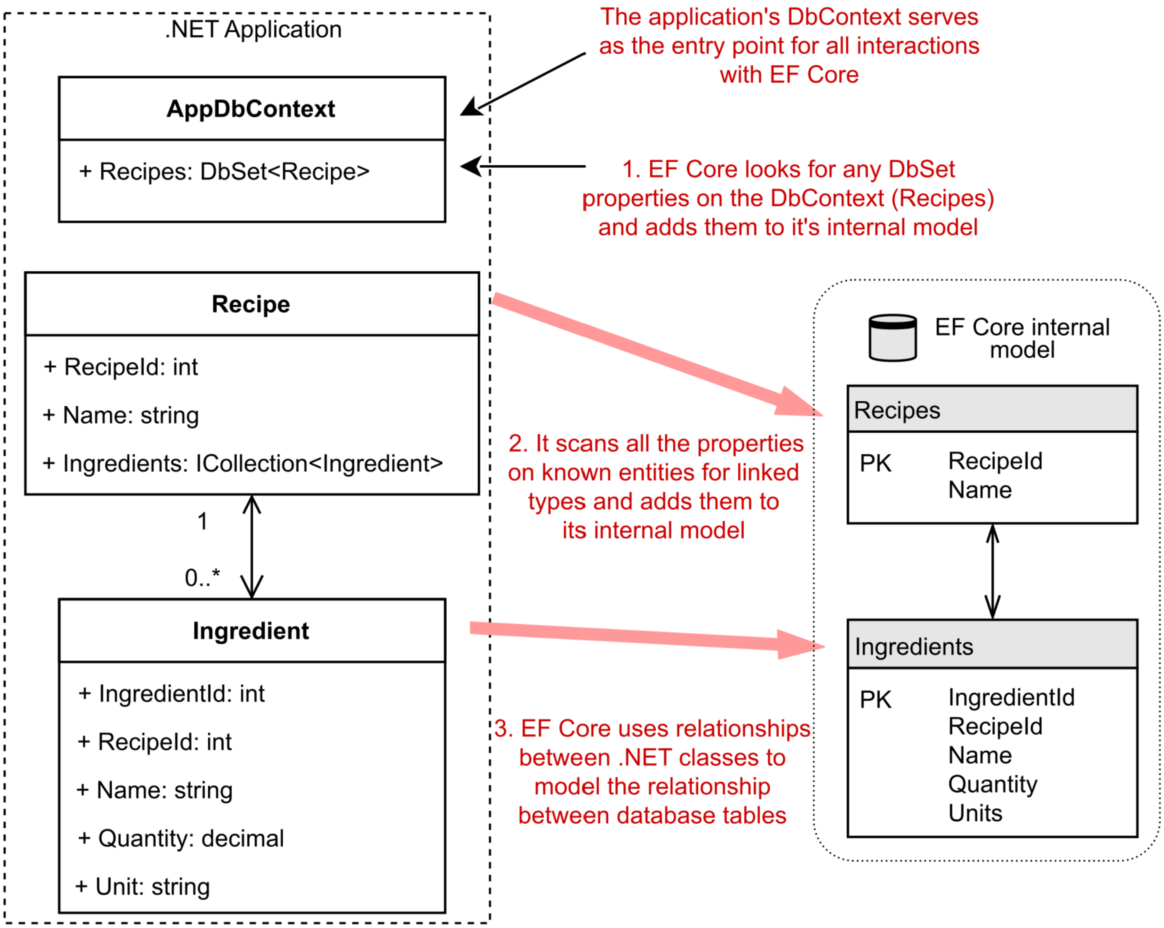

.NET Application
AppDbContext
+ Recipes: DbSet<Recipe>
Recipe
+ RecipeId: int
+ Name: string
+ Ingredients: ICollection<Ingredient>
1
0..*
Ingredient
+ IngredientId: int
+ RecipeId: int
+ Name: string
+ Quantity: decimal
+ Unit: string
The application's DbContext serves as the entry point for all interactions with EF Core
1. EF Core looks for any DbSet properties on the DbContext (Recipes) and adds them to it's internal model
2. It scans all the properties on known entities for linked types and adds them to its internal model
3. EF Core uses relationships between .NET classes to model the relationship between database tables
EF Core internal model
Recipes
PK
RecipeId
Name
Ingredients
PK
IngredientId
RecipeId
Name
Quantity
Units

Figure 12.3 EF Core creates an internal model of your application's data model by exploring the types in your code. It adds all of the types referenced in the DbSet<> properties on your app's DbContext, and any linked types.

For your recipe app, EF Core will build a model of the Recipe class because it's exposed on the AppDbContext as a DbSet<Recipe>. Furthermore, EF Core will loop through all the properties on Recipe, looking for types it doesn't know about, and add them to its internal model. In your app, the Ingredients collection on Recipe exposes the Ingredient entity as an ICollection<Ingredient>, so EF Core models the entity appropriately.

Each entity is mapped to a table in the database, but EF Core also maps the relationships between the entities. Each recipe can have *many* ingredients, but each ingredient (which has a name, quantity, and unit) belongs to *one* recipe, so this is a many-to-one relationship. EF Core uses that knowledge to correctly model the equivalent many-to-one database structure.

NOTE Two different recipes, say fish pie and lemon chicken, may use an ingredient that has both the same name and quantity, for example the juice of one lemon, but they're fundamentally two different instances. If you update the lemon chicken recipe to use two lemons, you wouldn't want this change to automatically update the fish pie to use two lemons too!

EF Core uses the internal model it builds when interacting with the database. This ensures it builds the correct SQL to create, read, update, and delete entities.

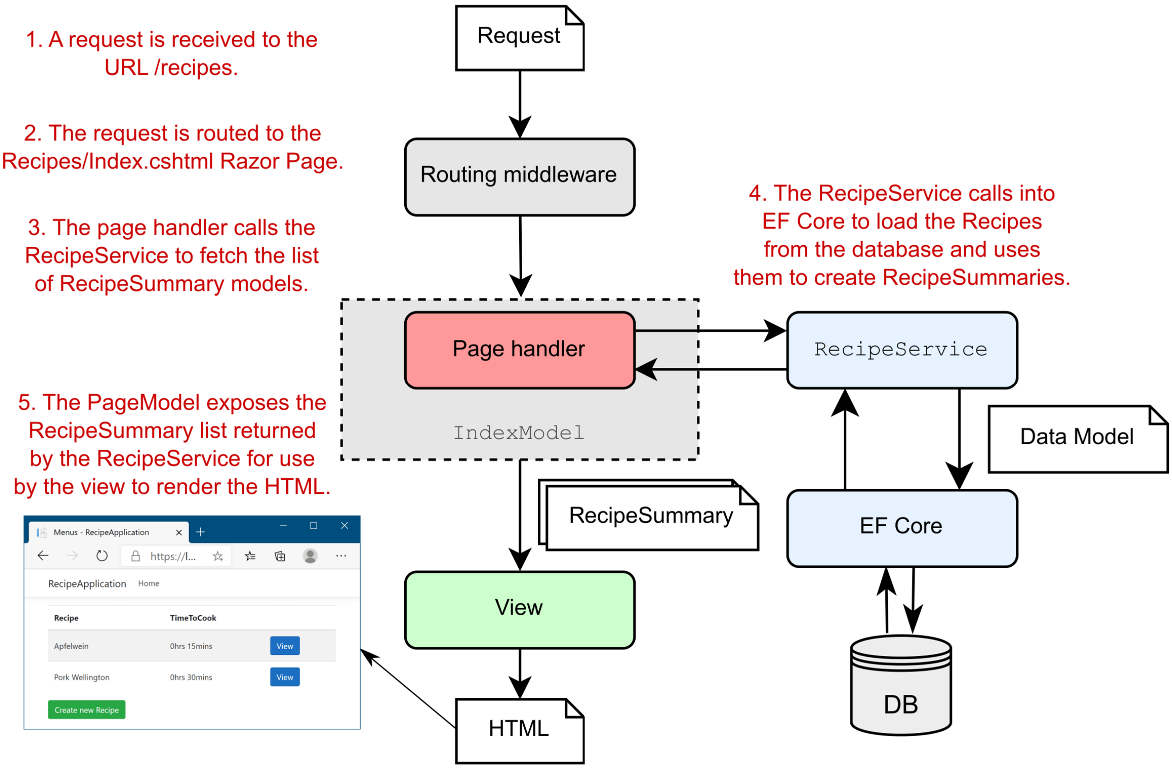
Right, it's about time for some code! In the next section, you'll start building the recipe app. You'll see how to add EF Core to an ASP.NET Core application, configure a database provider, and design your application's data model.

12.2 Adding EF Core to an application

In this section, we focus on getting EF Core installed and configured in your ASP.NET Core recipe app. You'll learn how to install the required NuGet packages and how to build the data model for your application. As we're talking about EF Core in this chapter, I'm not going to go into how to create the application in general—I created a simple Razor Pages app as the basis, nothing fancy.

TIP The sample code for this chapter shows the state of the application at three points in this chapter: the end of section 12.2, at the end of section 12.3, and at the end of the chapter. It also includes examples using both Local DB and SQLite providers.

Interaction with EF Core in the example app occurs in a service layer that encapsulates all the data access outside of the Razor Pages framework, as shown in figure 12.4. This keeps your concerns separated and makes your services testable.



Recipe	TimeToCook	
Apfelwein	0hrs 15mins	View
Pork Wellington	0hrs 30mins	View

1. A request is received to the URL /recipes.

2. The request is routed to the Recipes/Index.cshtml Razor Page.

3. The page handler calls the RecipeService to fetch the list of RecipeSummary models.

4. The RecipeService calls into EF Core to load the Recipes from the database and uses them to create RecipeSummaries.

5. The PageModel exposes the RecipeSummary list returned by the RecipeService for use by the view to render the HTML.

Figure 12.4 Handling a request by loading data from a database using EF Core. Interaction with EF Core is restricted to `RecipeService` only—the Razor Page doesn't access EF Core directly.

Adding EF Core to an application is a multistep process:

1. Choose a database provider; for example, Postgres, SQLite, or MS SQL Server.
2. Install the EF Core NuGet packages.
3. Design your app's `DbContext` and entities that make up your data model.
4. Register your app's `DbContext` with the ASP.NET Core DI container.
5. Use EF Core to generate a *migration* describing your data model.
6. Apply the migration to the database to update the database's schema.

This might seem a little daunting already, but we'll walk through steps 1–4 in this section, and steps 5–6 in section 12.3, so it won't take long. Given the space constraints of this chapter, I'm going to be sticking to the default conventions of EF Core in the code I show. EF Core is far more customizable than it may initially appear, but I encourage you to stick to the defaults wherever possible. It will make your life easier in the long run!

The first step in setting up EF Core is to decide which database you'd like to interact with. It's likely that a client or your company's policy will dictate this to you, but it's still worth giving the choice some thought.

12.2.1 Choosing a database provider and installing EF Core

EF Core supports a range of databases by using a provider model. The modular nature of EF Core means you can use the same high-level API to program against different underlying databases, and EF Core knows how to generate the necessary implementation-specific code and SQL statements.

You probably already have a database in mind when you start your application, and you'll be pleased to know that EF Core has got most of the popular ones covered. Adding support for a given database involves adding the correct NuGet package to your csproj file. For example:

- *PostgreSQL*—Npgsql.EntityFrameworkCore.PostgreSQL
- *Microsoft SQL Server*—Microsoft.EntityFrameworkCore.SqlServer
- *MySQL*—MySql.Data.EntityFrameworkCore
- *SQLite*—Microsoft.EntityFrameworkCore.SQLite

Some of the database provider packages are maintained by Microsoft, some are maintained by the open source community, and some may require a paid license (for example, the Oracle provider), so be sure to check your requirements. You can find a list of providers at https://docs.microsoft.com/ef/core/providers/.

You install a database provider into your application in the same way as any other library: by adding a NuGet package to your project's csproj file and running `dotnet restore` from the command line (or letting Visual Studio automatically restore for you).

EF Core is inherently modular, so you'll want to install two packages. I'm using the SQL Server database provider with LocalDB for the recipe app, so I'll be using the SQL Server packages.

- *Microsoft.EntityFrameworkCore.SqlServer*—This is the main database provider package for using EF Core at runtime. It also contains a reference to the main EF Core NuGet package.
- *Microsoft.EntityFrameworkCore.Design*—This contains shared design-time components for EF Core.

> **TIP** You'll also want to install tooling to help you create and update your database. I show how to install these in section 12.3.1.

Listing 12.1 shows the recipe app's csproj file after adding the EF Core packages. Remember, you add NuGet packages as `PackageReference` elements.

Listing 12.1 Installing EF Core into an ASP.NET Core application

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

```
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>                #A
  </PropertyGroup>

  <ItemGroup>
    <PackageReference                                            #B
        Include="Microsoft.EntityFrameworkCore.SqlServer"        #B
        Version="3.1.3" />                                       #B
    <PackageReference                                            #C
        Include="Microsoft.EntityFrameworkCore.Design"           #C
        Version="3.1.3" />                                       #C
  </ItemGroup>
</Project>
```

#A The app targets .NET Core 3.1.
#B Install the appropriate NuGet package for your selected DB
#C Contains shared design-time components for EF Core

With these packages installed and restored, you have everything you need to start building the data model for your application. In the next section, we'll create the entity classes and the `DbContext` for your recipe app.

12.2.2 Building a data model

In section 12.1.4, I showed an overview of how EF Core builds up its internal model of your database from the `DbContext` and entity models. Apart from this discovery mechanism, EF Core is pretty flexible in letting you define your entities the way *you* want to, as POCO classes.

Some ORMs require your entities to inherit from a specific base class or decorate your models with attributes to describe how to map them. EF Core heavily favors a convention over configuration approach, as you can see in this listing, which shows the `Recipe` and `Ingredient` entity classes for your app.

Listing 12.2 Defining the EF Core entity classes

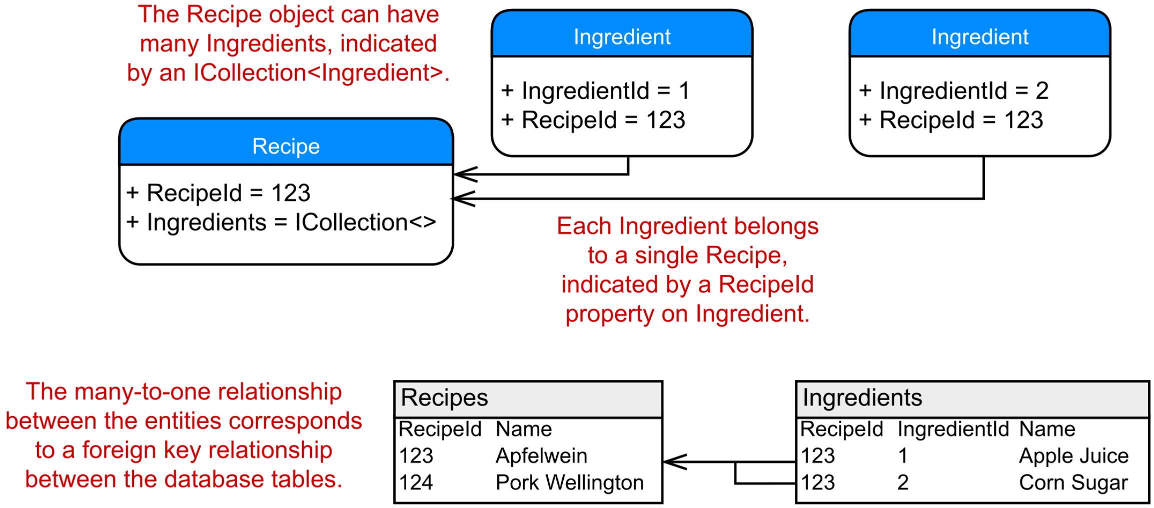
```
public class Recipe
{
    public int RecipeId { get; set; }
    public string Name { get; set; }
    public TimeSpan TimeToCook { get; set; }
    public bool IsDeleted { get; set; }
    public string Method { get; set; }
    public ICollection<Ingredient> Ingredients { get; set; }     #A
}
public class Ingredient
{
    public int IngredientId { get; set; }
    public int RecipeId { get; set; }
    public string Name { get; set; }
    public decimal Quantity { get; set; }
    public string Unit { get; set; }
}
```

#A A Recipe can have many Ingredients, represented by ICollection.

These classes conform to certain default conventions that EF Core uses to build up a picture of the database it's mapping. For example, the `Recipe` class has a `RecipeId` property and the `Ingredient` class has an `IngredientId` property. EF Core identifies this pattern of `TId` as indicating the *primary key* of the table.

DEFINITION **The *primary key* of a table is a value that uniquely identifies the row among all the others in the table. It's often an `int` or a `Guid`.**

Another convention visible here is the `RecipeId` property on the `Ingredient` class. EF Core interprets this to be a *foreign key* pointing to the `Recipe` class. When considered with `ICollection<Ingredient>` on the `Recipe` class, this represents a many-to-one relationship, where each recipe has many ingredients, but each ingredient only belongs to a single recipe, as shown in figure 12.5.



The Recipe object can have many Ingredients, indicated by an ICollection<Ingredient>.

Each Ingredient belongs to a single Recipe, indicated by a RecipeId property on Ingredient.

The many-to-one relationship between the entities corresponds to a foreign key relationship between the database tables.

Recipes

RecipeId	Name
123	Apfelwein
124	Pork Wellington

Ingredients

RecipeId	IngredientId	Name
123	1	Apple Juice
123	2	Corn Sugar

Figure 12.5 Many-to-one relationships in code are translated to foreign key relationships between tables.

DEFINITION **A *foreign key* on a table points to the primary key of a different table, forming a link between the two rows.**

Many other conventions are at play here, such as the names EF Core will assume for the database tables and columns, or the database column types it will use for each property, but I'm not going to discuss them here. The EF Core documentation contains details about all of the conventions, as well as how to customize them for your application: https://docs.microsoft.com/ef/core/modeling/.

> **TIP** You can also use `DataAnnotations` attributes to decorate your entity classes, controlling things like column naming or `string` length. EF Core will use these attributes to override the default conventions.

As well as the entities, you also define the `DbContext` for your application. This is the heart of EF Core in your application, used for all your database calls. Create a custom `DbContext`, in this case called `AppDbContext`, and derive from the `DbContext` base class, as shown next. This exposes the `DbSet<Recipe>` so EF Core can discover and map the `Recipe` entity. You can expose multiple instances of `DbSet<>` in this way, for each of the "top-level" entities in your application.

Listing 12.3 Defining the application `DbContext`

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)    #A
        : base(options) { }                                        #A
    public DbSet<Recipe> Recipes { get; set; }                     #B
}
```

#A The constructor options object, containing details such as the connection string
#B You'll use the Recipes property to query the database.

The `AppDbContext` for your app is simple, containing a list of your root entities, but you can do a lot more with it in a more complex application. If you wanted, you could completely customize how EF Core maps entities to the database, but for this app you're going to use the defaults.

> **NOTE** You didn't list `Ingredient` on `AppDbContext`, but it will be modeled by EF Core as it's exposed on the `Recipe`. You can still access the `Ingredient` objects in the database, but you have to go *via* the `Recipe` entity's `Ingredients` property to do so, as you'll see in section 12.4.

For this simple example, your data model consists of these three classes: `AppDbContext`, `Recipe`, and `Ingredient`. The two entities will be mapped to tables and their columns to properties, and you'll use the `AppDbContext` to access them.

> **NOTE** This *code first* approach is typical, but if you have an existing database, you can automatically generate the EF entities and `DbContext` instead. (More information can be found at https://docs.microsoft.com/ef/core/managing-schemas/scaffolding.)

The data model is complete, but you're not quite ready to use it yet. Your ASP.NET Core app doesn't know how to create your `AppDbContext`, and your `AppDbContext` needs a connection string so that it can talk to the database. In the next section, we'll tackle both of these issues, and will finish setting up EF Core in your ASP.NET Core app.

12.2.3 Registering a data context

Like any other service in ASP.Net Core, you should register your AppDbContext with the DI container. When registering your context, you also configure the database provider and set the connection string, so EF Core knows how to talk with the database.

You register the AppDbContext in the ConfigureServices method of Startup.cs. EF Core provides a generic AddDbContext<T> extension method for this purpose, which takes a configuration function for a DbContextOptionsBuilder instance. This builder can be used to set a host of internal properties of EF Core and lets you completely replace the internal services of EF Core if you want.

The configuration for your app is, again, nice and simple, as you can see in the following listing. You set the database provider with the UseSqlServer extension method, made available by the Microsoft.EntityFrameworkCore.SqlServer package, and pass it a connection string.

Listing 12.4 Registering a DbContext with the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    var connString = Configuration                         #A
        .GetConnectionString("DefaultConnection");          #A

    services.AddDbContext<AppDbContext>(                    #B
        options => options.UseSqlServer(connString));       #C

    // Add other services.
}
```

#A The connection string is taken from configuration, from the ConnectionStrings section.
#B Register your app's DbContext by using it as the generic parameter.
#C Specify the database provider in the customization options for the DbContext.

> **NOTE** **If you're using a different database provider, for example a provider for SQLite, you will need to call the appropriate** Use* **method on the** options **object when registering your** AppDbContext.

The connection string is a typical secret as I discussed in the previous chapter, so loading it from configuration makes sense. At runtime, the correct configuration string for your current environment will be used, so you can use different databases when developing locally and in production.

> **TIP** **You can configure your** AppDbContext **in other ways and provide the connection string, such as with the** OnConfiguring **method, but I recommend the method shown here for ASP.NET Core websites.**

You now have a DbContext, AppDbContext, registered with the DI container, and a data model corresponding to your database. Code-wise, you're ready to start using EF Core, but the one thing you *don't* have is a database! In the next section, you'll see how you can easily use the .NET CLI to ensure your database stays up to date with your EF Core data model.

12.3 Managing changes with migrations

In this section, you'll learn how to generate SQL statements to keep your database's schema in sync with your application's data model, using migrations. You'll learn how to create an initial migration and use it to create the database. You'll then update your data model, create a second migration, and use it to update the database schema.

Managing *schema* changes for databases, such as when you need to add a new table or a new column, is notoriously difficult. Your application code is explicitly tied to a particular *version* of a database, and you need to make sure the two are always in sync.

> **DEFINITION** *Schema* refers to how the data is organized in a database, including, among others things, the tables, columns, and the relationships between them.

When you deploy an app, you can normally delete the old code/executable and replace it with the new code—job done. If you need to roll back a change, delete that new code and deploy an old version of the app.

The difficulty with databases is that they contain data! That means that blowing it away and creating a new database with every deployment isn't possible.

A common best practice is to explicitly version a database's schema along with your application's code. You can do this in a number of ways but, typically, you need to store a diff between the previous schema of the database and the new schema, often as a SQL script. You can then use libraries such as DbUp and FluentMigrator[55] to keep track of which scripts have been applied and ensure your database schema is up to date. Alternatively, you can use external tools that manage this for you.

EF Core provides its own version of schema management called *migrations*. Migrations provide a way to manage changes to a database schema when your EF Core data model changes. A migration is a C# code file in your application that defines how the data model changed—which columns were added, new entities, and so on. Migrations provide a record over time of how your database schema evolved as part of your application, so the schema is always in sync with your app's data model.

You can use command-line tools to create a new database from the migrations, or to update an existing database by *applying* new migrations to it. You can even rollback a migration, which will update a database to a previous schema.

> **WARNING** Applying migrations modifies the database, so you always have to be aware of data loss. If you remove a table from the database using a migration and then rollback the migration, the table will be recreated, but the data it previously contained will be gone forever!

[55] DbUp and FluentMigrator are open source projects, available at https://github.com/fluentmigrator/fluentmigrator and https://github.com/DbUp/DbUp respectively.

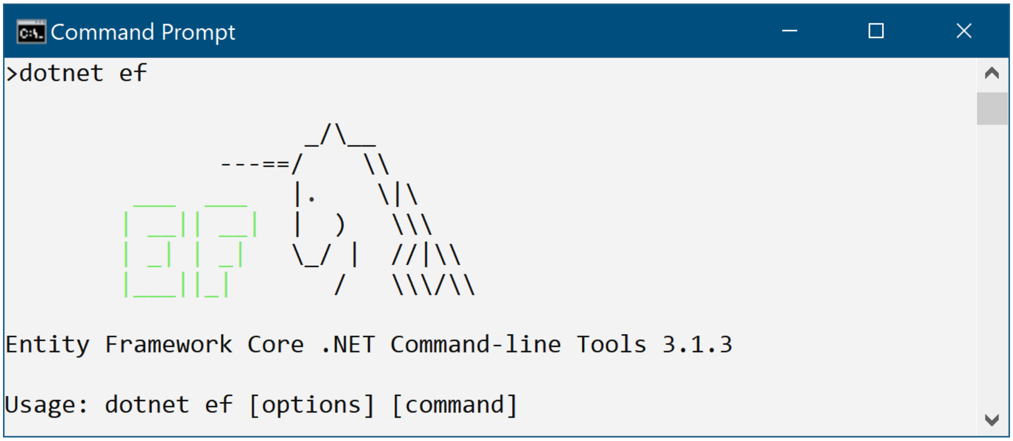
In this section, you'll see how to create your first migration and use it to create a database. You'll then update your data model, create a second migration, and use it to update the database schema.

12.3.1 Creating your first migration

Before you can create migrations, you'll need to install the necessary tooling. There are two primary ways to do this:

- *Package manager console*—Provides a number of PowerShell cmdlets for use inside Visual Studio's Package Manager Console (PMC). You can install them directly from the PMC or by adding the Microsoft.EntityFrameworkCore.Tools package to your project.
- *.NET Core tool*—Cross-platform tooling that you can run from the command line which extends the .NET SDK. You can install these tools globally for your machine by running `dotnet tool install --global dotnet-ef`.[56]

In this book, I'll be using the cross-platform .NET Core tools, but if you're familiar with EF 6.X or prefer to use the Visual Studio PMC, then there are equivalent commands for all of the steps you're going to take.[57] You can check the .NET Core tool installed correctly by running `dotnet ef`. This should produce a help screen like the one shown in figure 12.6.



```
>dotnet ef

                     _/\__
               ---==/    \\
         ___  ___   |.    \|\
        | __|| __|  |  )    \\\
        | _| | _|   \_/ |  //|\\
        |___||_|       /   \\\/\\

Entity Framework Core .NET Command-line Tools 3.1.3

Usage: dotnet ef [options] [command]
```

Figure 12.6 Running the `dotnet ef` command to check the .NET Core EF Core tools are installed correctly.

[56] Alternatively, you can install the tools *locally*, using a *tool manifest file*. To see how to use this approach, see https://andrewlock.net/new-in-net-core-3-local-tools/.

[57] Documentation for PowerShell cmdlets can be found at https://docs.microsoft.com/ef/core/miscellaneous/cli/powershell.

> **TIP** If you get the *No executable found matching command "dotnet-ef"* message when running the preceding command, make sure you have installed the global tool using `dotnet tool install --global dotnet-ef`. In general, you need to run the `dotnet ef` tools from the *project* folder in which you have registered your `AppDbContext` (not at the solution-folder level).

With the tools installed and your database context configured, you can create your first migration by running the following command from inside your web project folder and providing a name for the migration—in this case, "`InitialSchema`":

```
dotnet ef migrations add InitialSchema
```

This command creates three files in the Migrations folder in your project:

- *Migration file*—A file with the Timestamp_MigrationName.cs format. This describes the actions to take on the database, such as Create table or Add column. Note the commands generated here are *database provider specific*, based on the database provider configured in your project.
- *Migration designer.cs file*—This file describes EF Core's internal model of your data model at the point in time the migration was generated.
- *AppDbContextModelSnapshot.cs*—This describes EF Core's *current* internal model. This will be updated when you add another migration, so it should always be the same as the current, latest migration.

  EF Core can use AppDbContextModelSnapshot.cs to determine a database's previous state when creating a new migration, without interacting with the database directly.

These three files encapsulate the migration process but adding a migration doesn't update anything in the database itself. For that, you must run a different command to apply the migration to the database.

> **TIP** You can, and should, look inside the migration file EF Core generates to check what it will do to your database before running the following commands. Better safe than sorry!

You can apply migrations in one of three ways:

- Using the .NET Core tool
- Using the Visual Studio PowerShell cmdlets
- In code, by obtaining an instance of your `AppDbContext` from the DI container and calling `context.Database.Migrate()`.

Which is best for you is a matter of how you've designed your application, how you'll update your production database, and your personal preference. I'll use the .NET Core tool for now, but I discuss some of these considerations in section 12.5.

You can apply migrations to a database by running

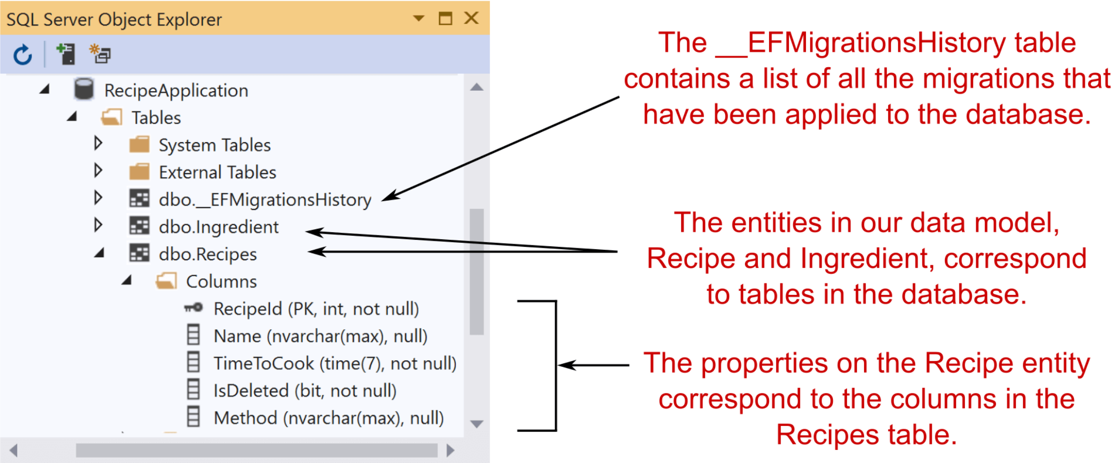
```
dotnet ef database update
```

from the project folder of your application. I won't go into the details of how this works, but this command performs four steps:

1. Builds your application.
2. Loads the services configured in your app's `Startup` class, including `AppDbContext`.
3. Checks whether the database in the `AppDbContext` connection string exists. If not, creates it.
4. Updates the database by applying any unapplied migrations.

If everything is configured correctly, as I showed in section 12.2, then running this command will set you up with a shiny new database, such as the one shown in figure 12.7!

> **REMINDER** If you get an error message "No project was found" when running these commands, check that you're running them in your application's *project* folder, not the top-level solution folder.



SQL Server Object Explorer

- RecipeApplication
  - Tables
    - System Tables
    - External Tables
    - dbo.__EFMigrationsHistory
    - dbo.Ingredient
    - dbo.Recipes
      - Columns
        - RecipeId (PK, int, not null)
        - Name (nvarchar(max), null)
        - TimeToCook (time(7), not null)
        - IsDeleted (bit, not null)
        - Method (nvarchar(max), null)

The __EFMigrationsHistory table contains a list of all the migrations that have been applied to the database.

The entities in our data model, Recipe and Ingredient, correspond to tables in the database.

The properties on the Recipe entity correspond to the columns in the Recipes table.

Figure 12.7 Applying migrations to a database will create the database if it doesn't exist and update the database to match EF Core's internal data model. The list of applied migrations is stored in the __EFMigrationsHistory table.

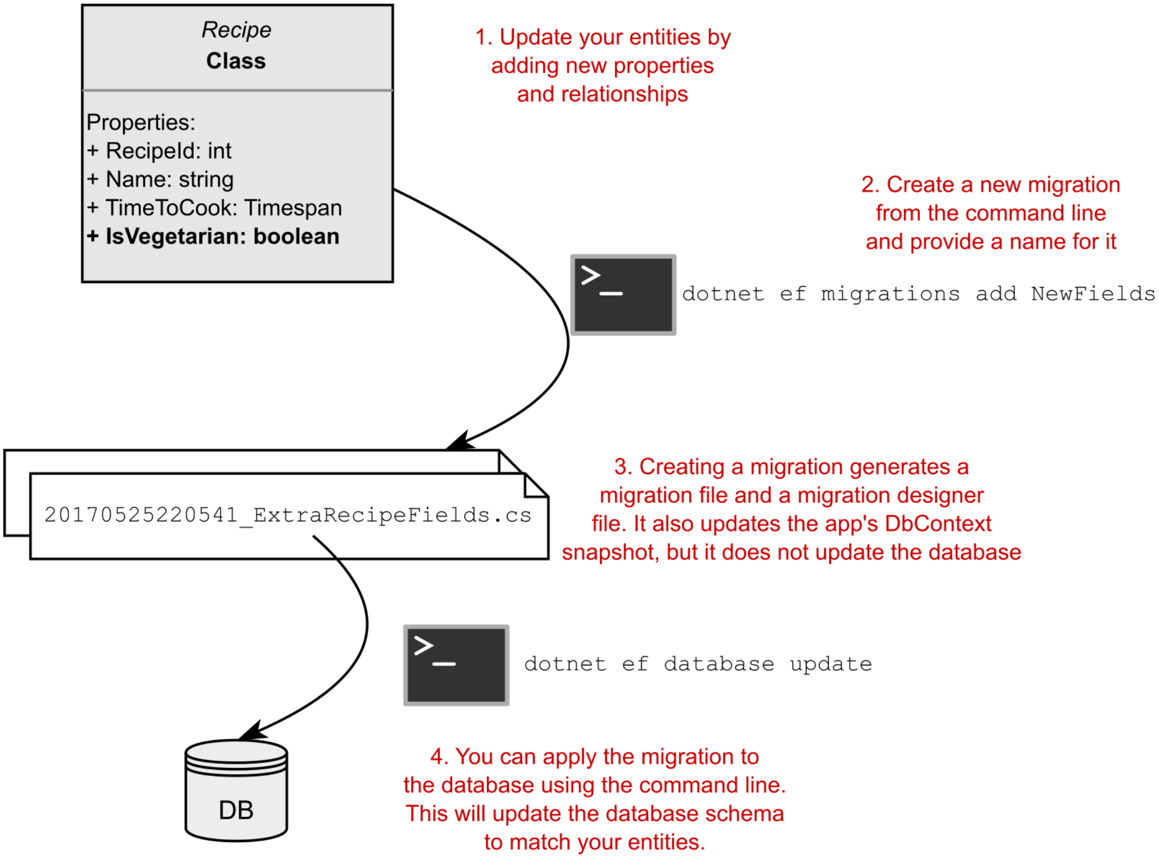
When you apply the migrations to the database, EF Core creates the necessary tables in the database and adds the appropriate columns and keys. You may have also noticed the __EFMigrationsHistory table. EF Core uses this to store the names of migrations that it's applied to the database. Next time you run `dotnet ef database update`, EF Core can compare this table to the list of migrations in your app and will apply only the new ones to your database.

In the next section, we'll look at how this makes it easy to change your data model, and update the database schema, without having to recreate the database from scratch.

12.3.2 Adding a second migration

Most applications inevitably evolve, whether due to increased scope or simple maintenance. Adding properties to your entities, adding new entities entirely, and removing obsolete classes—all are likely.

EF Core migrations make this simple. Change your entities to your desired state, generate a migration, and apply it to the database, as shown in figure 12.8. Imagine you decide that you'd like to highlight vegetarian and vegan dishes in your recipe app by exposing `IsVegetarian` and `IsVegan` properties on the `Recipe` entity.


Recipe
Class
Properties:
+ RecipeId: int
+ Name: string
+ TimeToCook: Timespan
+ IsVegetarian: boolean
1. Update your entities by adding new properties and relationships
2. Create a new migration from the command line and provide a name for it
>_
20170525220541_ExtraRecipeFields.cs
3. Creating a migration generates a migration file and a migration designer file. It also updates the app's DbContext snapshot, but it does not update the database
>_
DB
4. You can apply the migration to the database using the command line. This will update the database schema to match your entities.

```
dotnet ef migrations add NewFields
```

```
dotnet ef database update
```

Figure 12.8 Creating a second migration and applying it to the database using the command-line tools.

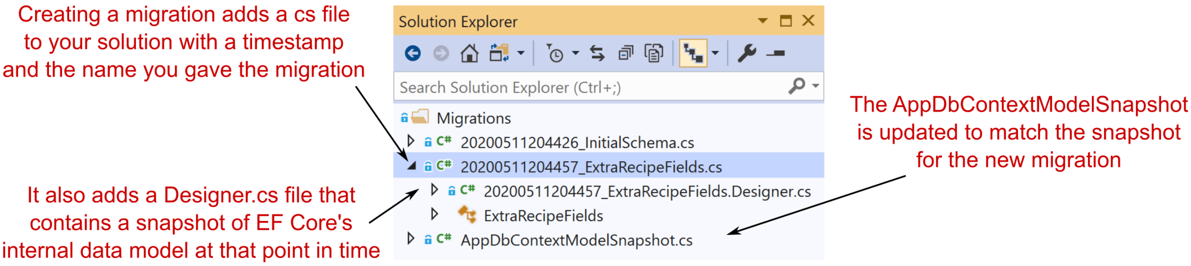
Listing 12.5 Adding properties to the `Recipe` entity

```
public class Recipe
{
    public int RecipeId { get; set; }
    public string Name { get; set; }
    public TimeSpan TimeToCook { get; set; }
    public bool IsDeleted { get; set; }
    public string Method { get; set; }
    public bool IsVegetarian { get; set; }
    public bool IsVegan { get; set; }
    public ICollection<Ingredient> Ingredients { get; set; }
}
```

After changing your entities, you need to update EF Core’s internal representation of your data model. You do this in the exact same way as for the first migration, by calling `dotnet ef migrations add` and providing a name for the migration:

```
dotnet ef migrations add ExtraRecipeFields
```

This creates a second migration in your project by adding the migration file and its .designer.cs snapshot file and updating AppDbContextModelSnapshot.cs, as shown in figure 12.9.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Migrations
20200511204426_InitialSchema.cs
20200511204457_ExtraRecipeFields.cs
20200511204457_ExtraRecipeFields.Designer.cs
ExtraRecipeFields
AppDbContextModelSnapshot.cs

Creating a migration adds a cs file to your solution with a timestamp and the name you gave the migration

It also adds a Designer.cs file that contains a snapshot of EF Core's internal data model at that point in time

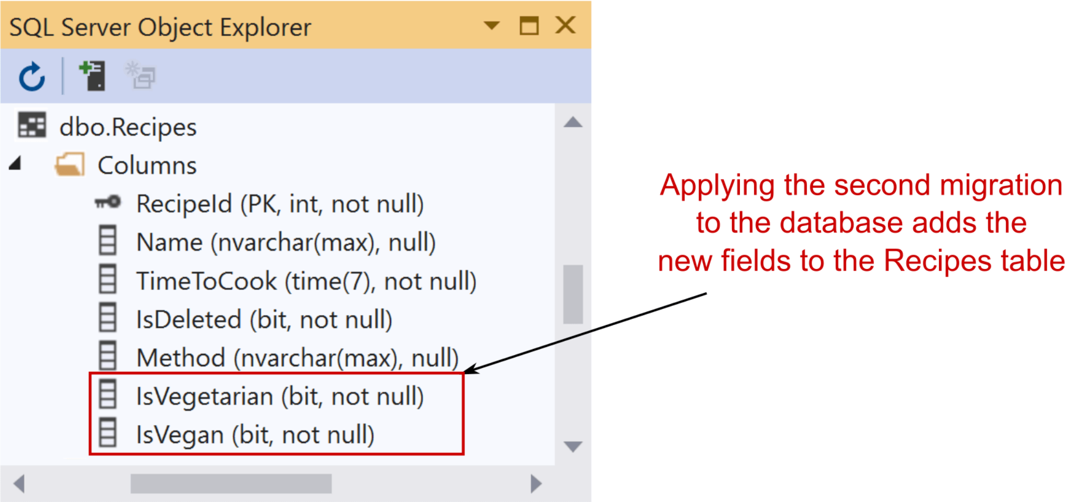
The AppDbContextModelSnapshot is updated to match the snapshot for the new migration

Figure 12.9 Adding a second migration adds a new migration file and a migration Designer.cs file. It also updates AppDbContextModelSnapshot to match the new migration’s Designer.cs file.

As before, this creates the migration’s files, but it doesn’t modify the database. You can apply the migration, and update the database, by running

```
dotnet ef database update
```

This compares the migrations in your application to the __EFMigrationsHistory table on your database to see which migrations are outstanding and then runs them. EF Core will run the 20200511204457_ExtraRecipeFields migration, adding the `IsVegetarian` and `IsVegan` fields to the database, as shown in figure 12.10.



SQL Server Object Explorer

dbo.Recipes
Columns
RecipeId (PK, int, not null)
Name (nvarchar(max), null)
TimeToCook (time(7), not null)
IsDeleted (bit, not null)
Method (nvarchar(max), null)
IsVegetarian (bit, not null)
IsVegan (bit, not null)

Applying the second migration to the database adds the new fields to the Recipes table

Figure 12.10 Applying the ExtraRecipeFields migration to the database adds the `IsVegetarian` and `IsVegan` fields to the Recipes table.

Using migrations is a great way to ensure your database is versioned along with your app code in source control. You can easily check out your app's source code for a historical point in time and recreate the database schema that your application used at that point.

Migrations are easy to use when you're working alone, or when you're deploying to a single web server, but even in these cases, there are important things to consider when deciding how to manage your databases. For apps with multiple web servers using a shared database, or for containerized applications, you have even more things to think about.

This book is about ASP.NET Core, not EF Core, so I don't want to dwell on database management too much, but section 12.5 points out some of the things you need to bear in mind when using migrations in production.

In the next section, we'll get back to the meaty stuff—defining our business logic and performing CRUD operations on the database.

12.4 Querying data from and saving data to the database

Let's review where you are in creating the recipe application:

- You created a simple data model for the application, consisting of recipes and ingredients.
- You generated migrations for the data model, to update EF Core's internal model of your entities.
- You applied the migrations to the database, so its schema matches EF Core's model.

In this section, you'll build the business logic for your application by creating a `RecipeService`. This handles querying the database for recipes, creating new recipes, and modifying existing ones. As this app only has a simple domain, I'll be using `RecipeService` to handle all the requirements, but in your own apps you may have multiple services that cooperate to provide the business logic.

> **NOTE** For simple apps, you may be tempted to move this logic into your Razor Pages. I'd encourage you to resist this urge; extracting your business logic to other services decouples the HTTP-centric nature of Razor Pages and Web APIs from the underlying business logic. This will often make your business logic easier to test and more reusable.

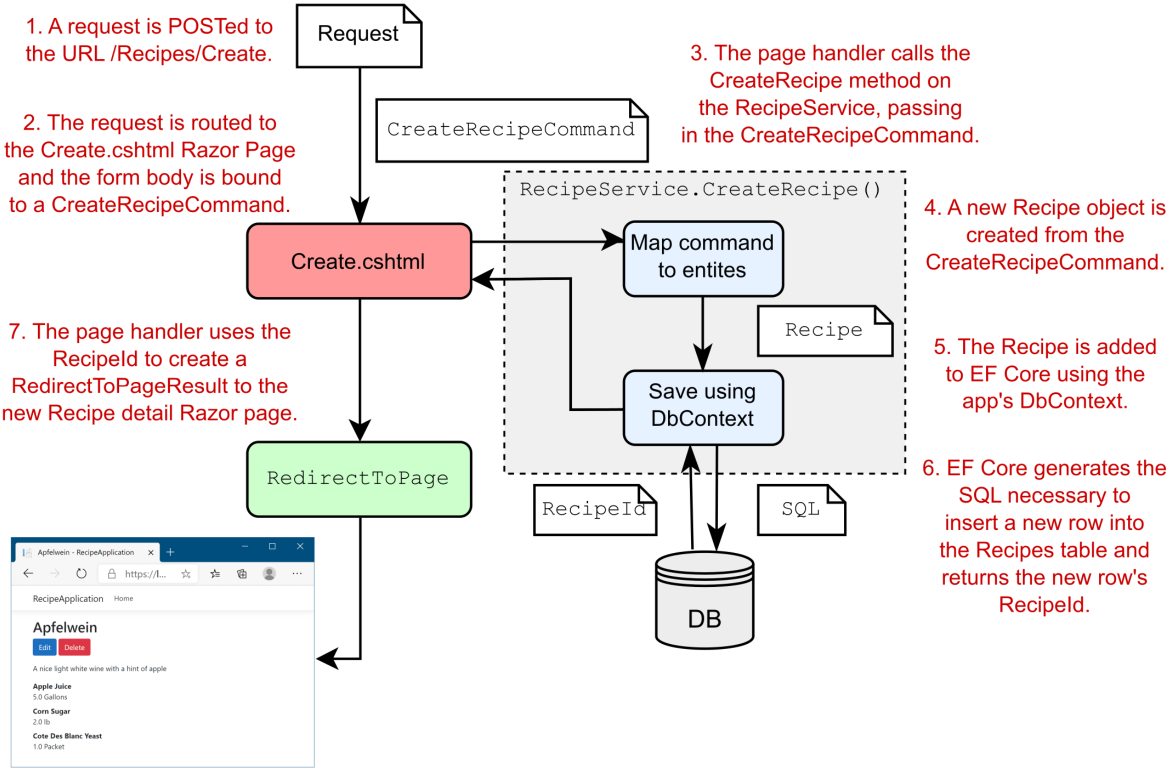
Our database doesn't have any data in it yet, so we'd better start by letting you create a recipe!

12.4.1 Creating a record

In this section, you're going to build the functionality to let users create a recipe in the app. This will primarily consist of a form that the user can use to enter all the details of the recipe using Razor Tag Helpers, which you learned about in chapters 7 and 8. This form is posted to the Create.cshtml Razor Page, which uses model binding and validation attributes to confirm the request is valid, as you saw in chapter 6.

If the request is valid, the page handler calls `RecipeService` to create the new `Recipe` object in the database. As EF Core is the topic of this chapter, I'm going to focus on this service alone, but you can always check out the source code for this book if you want to see how everything fits together.

The business logic for creating a recipe in this application is simple—there *is* no logic! Map the *command* binding model provided in the Create.cshtml Razor Page to a `Recipe` entity and its `Ingredients`, add the `Recipe` object to `AppDbContext`, and save that in the database, as shown in figure 12.11.



1. A request is POSTed to the URL /Recipes/Create.

2. The request is routed to the Create.cshtml Razor Page and the form body is bound to a CreateRecipeCommand.

3. The page handler calls the CreateRecipe method on the RecipeService, passing in the CreateRecipeCommand.

4. A new Recipe object is created from the CreateRecipeCommand.

5. The Recipe is added to EF Core using the app's DbContext.

6. EF Core generates the SQL necessary to insert a new row into the Recipes table and returns the new row's RecipeId.

7. The page handler uses the RecipeId to create a RedirectToPageResult to the new Recipe detail Razor page.

Apfelwein - RecipeApplication

RecipeApplication Home

Apfelwein

Edit Delete

A nice light white wine with a hint of apple

Apple Juice
5.0 Gallons

Corn Sugar
2.0 lb

Cote Des Blanc Yeast
1.0 Packet

Figure 12.11 Calling the Create.cshtml Razor Page and creating a new entity. A `Recipe` is created from the `CreateRecipeCommand` binding model and is added to the `DbContext`. EF Core generates the SQL to add a new row to the Recipes table in the database.

> **WARNING** Many simple, equivalent, sample applications using EF or EF Core allow you to bind *directly* to the `Recipe` entity as the view model for your MVC actions. Unfortunately, this exposes a security vulnerability known as overposting and is a bad practice. If you want to avoid the boilerplate mapping code in your applications, consider using a library such as AutoMapper (http://automapper.org/). For more details on overposting, see https://andrewlock.net/preventing-mass-assignment-or-over-posting-with-razor-pages-in-asp-net-core/.

Creating an entity in EF Core involves adding a new row to the mapped table. For your application, whenever you create a new `Recipe`, you also add the linked `Ingredient` entities. EF Core takes care of linking these all correctly by creating the correct `RecipeId` for each `Ingredient` in the database.

The bulk of the code required for this example involves translating from `CreateRecipeCommand` to the `Recipe` entity—the interaction with the `AppDbContext` consists of only two methods: `Add()` and `SaveChangesAsync()`.

Listing 12.6 Creating a `Recipe` entity in the database

```
readonly AppDbContext _context;                      #A
public async Task<int> CreateRecipe(CreateRecipeCommand cmd)     #B
{
    var recipe = new Recipe                              #C
    {                                                    #C
        Name = cmd.Name,                                 #C
        TimeToCook = new TimeSpan(                       #C
            cmd.TimeToCookHrs, cmd.TimeToCookMins, 0),   #C
        Method = cmd.Method,                             #C
        IsVegetarian = cmd.IsVegetarian,                 #C
        IsVegan = cmd.IsVegan,                           #C
        Ingredients = cmd.Ingredients?.Select(i =>       #C
        new Ingredient                      #D
        {                                   #D
            Name = i.Name,                  #D
            Quantity = i.Quantity,          #D
            Unit = i.Unit,                  #D
        }).ToList()                         #D
    };
    _context.Add(recipe);         #E
    await _context.SaveChangesAsync();       #F
    return recipe.RecipeId;       #G
}
```

#A An instance of the AppDbContext is injected in the class constructor using DI.
#B CreateRecipeCommand is passed in from the Razor Page handler.
#C Create a Recipe by mapping from the command object to the Recipe entity.
#D Map each CreateIngredientCommand onto an Ingredient entity.
#E Tell EF Core to track the new entities.
#F Tell EF Core to write the entities to the database. This uses the async version of the command.
#G EF Core populates the RecipeId field on your new Recipe when it's saved.

All interactions with EF Core and the database start with an instance of `AppDbContext`, which is typically DI injected via the constructor. Creating a new entity requires three steps:

1. Create the `Recipe` and `Ingredient` entities.
2. Add the entities to EF Core's list of tracked entities using `_context.Add(entity)`.
3. Execute the SQL `INSERT` statements against the database, adding the necessary rows to the `Recipe` and `Ingredient` tables, by calling `_context.SaveChangesAsync()`.

> **TIP** **There are *sync* and *async* versions of most of the EF Core commands that involve interacting with the database, such as `SaveChanges()` and `SaveChangesAsync()`. In general, the async versions will allow your app to handle more concurrent connections, so I tend to favor them whenever I can use them.**

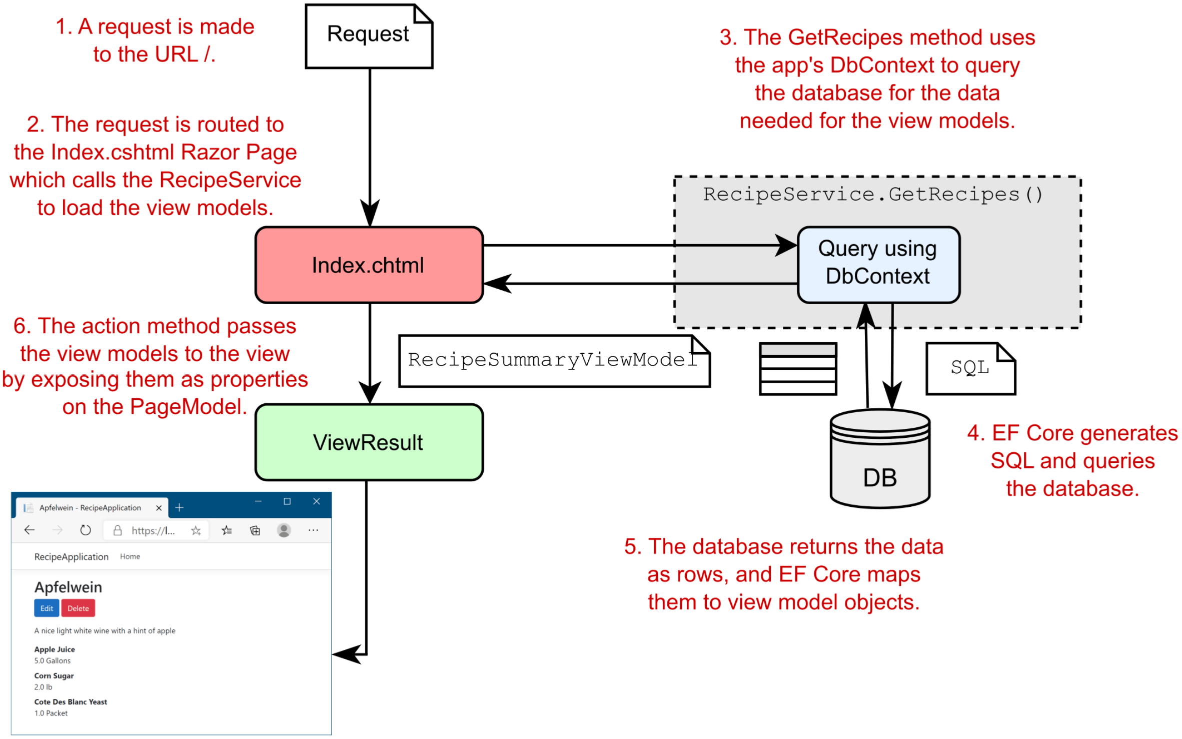
If there's a problem when EF Core tries to interact with your database—you haven't run the migrations to update the database schema, for example—it will throw an exception. I haven't shown it here, but it's important to handle these in your application, so you don't present users with an ugly error page when things go wrong.

Assuming all goes well, EF Core updates all the auto-generated IDs of your entities (RecipeId on Recipe, and both RecipeId and IngredientId on Ingredient). Return the recipe ID to the Razor Page so it can use it, for example, to redirect to the View Recipe page.

And there you have it—you've created your first entity using EF Core. In the next section, we'll look at loading these entities from the database so you can view them in a list.

12.4.2 Loading a list of records

Now that you can create recipes, you need to write the code to view them. Luckily, loading data is simple in EF Core, relying heavily on LINQ methods to control which fields you need. For your app, you'll create a method on RecipeService that returns a summary view of a recipe, consisting of the RecipeId, Name, and TimeToCook as a RecipeSummaryViewModel, as shown in figure 12.12.



1. A request is made to the URL /.

2. The request is routed to the Index.cshtml Razor Page which calls the RecipeService to load the view models.

3. The GetRecipes method uses the app's DbContext to query the database for the data needed for the view models.

4. EF Core generates SQL and queries the database.

5. The database returns the data as rows, and EF Core maps them to view model objects.

6. The action method passes the view models to the view by exposing them as properties on the PageModel.

Apfelwein

A nice light white wine with a hint of apple

Apple Juice
5.0 Gallons

Corn Sugar
2.0 lb

Cote Des Blanc Yeast
1.0 Packet

Figure 12.12 Calling the Index.cshtml Razor Page and querying the database to retrieve a list of RecipeSummaryViewModels. EF Core generates the SQL to retrieve the necessary fields from the database and maps them to view model objects.

NOTE Creating a view model is technically a UI concern rather than a business logic concern. I'm returning them directly from `RecipeService` here mostly to hammer home that you shouldn't be using EF Core entities directly in your Razor Pages.

The `GetRecipes` method in `RecipeService` is conceptually simple and follows a common pattern for querying an EF Core database, as shown in figure 12.13.

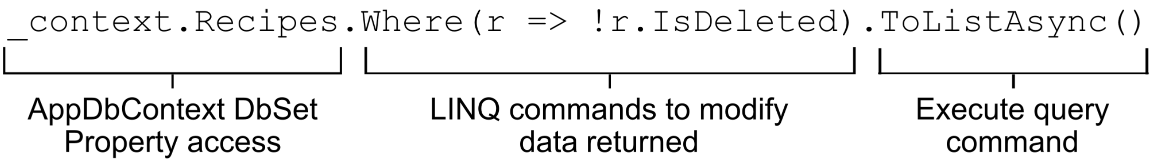

_context.Recipes.Where(r => !r.IsDeleted).ToListAsync()
AppDbContext DbSet Property access
LINQ commands to modify data returned
Execute query command

Figure 12.13 The three parts of an EF Core database query.

EF Core uses a fluent chain of LINQ commands to define the query to return on the database. The `DbSet<Recipe>` property on `AppDataContext` is an `IQueryable`, so you can use all the usual `Select()` and `Where()` clauses that you would with other `IQueryable` providers. EF Core will convert these into a SQL statement to query the database with when you call an *execute* function such as `ToListAsync()`, `ToArrayAsync()`, or `SingleAsync()`, or their non-async brethren.

You can also use the `Select()` extension method to map to objects other than your entities as part of the SQL query. You can use this to efficiently query the database by only fetching the columns you need.

Listing 12.7 shows the code to fetch a list of `RecipeSummaryViewModel`s, following the same basic pattern as in figure 12.12. It uses a `Where` LINQ expression to filter out recipes marked as deleted, and a `Select` clause to map to the view models. The `ToListAsync()` command instructs EF Core to generate the SQL query, execute it on the database, and build `RecipeSummaryViewModel` from the data returned.

Listing 12.7 Loading a list of items using EF Core

```
public async Task<ICollection<RecipeSummaryViewModel>> GetRecipes()
{
    return await _context.Recipes                       #A
        .Where(r => !r.IsDeleted)
        .Select(r => new RecipeSummaryViewModel                #B
        {                                                      #B
            Id = r.RecipeId,                                   #B
            Name = r.Name,                                     #B
            TimeToCook = "{r.TimeToCook.TotalMinutes}mins"     #B
        })
        .ToListAsync();              #C
}
```

#A A query starts from a DbSet property
#B EF Core will only query the Recipe columns it needs to map the view model correctly

#C This executes the SQL query and creates the final view models.

Notice that in the `Select` method, you convert the `TimeToCook` property from a `TimeSpan` to a `string` using string interpolation:

```
TimeToCook = $"{x.TimeToCook.TotalMinutes}mins"
```

I said before that EF Core converts the series of LINQ expressions into SQL, but that's only a half-truth; EF Core can't or doesn't know how to convert some expressions to SQL. For those cases, such as in this example, EF Core finds the fields from the DB that it needs in order to run the expression on the client side, selects those from the database, and then runs the expression in C# afterwards. This lets you combine the power and performance of database-side evaluation without compromising the functionality of C#.

> **WARNING** **Client-side evaluation is both powerful and useful but has the potential to cause issues. In general, recent versions of EF Core will throw an exception if a query requires dangerous client-side evaluation. For examples, including how to avoid these issues, see the documentation at https://docs.microsoft.com/ef/core/querying/client-eval.**

At this point, you have a list of records, displaying a summary of the recipe's data, so the obvious next step is to load the detail for a single record.

12.4.3 Loading a single record

For most intents and purposes, loading a single record is the same as loading a list of records. They share the same common structure you saw in figure 12.13, but when loading a single record, you'll typically use a `Where` clause and execute a command that restricts the data to a single entity.

Listing 12.8 shows the code to fetch a recipe by ID following the same basic pattern as before (figure 12.12). It uses a `Where()` LINQ expression to restrict the query to a single recipe, where `RecipeId == id`, and a `Select` clause to map to `RecipeDetailViewModel`. The `SingleOrDefaultAsync()` clause will cause EF Core to generate the SQL query, execute it on the database, and build the view model.

> **NOTE** `SingleOrDefaultAsync()` **will throw an exception if the previous** `Where` **clause returns more than one record.**

Listing 12.8 Loading a single item using EF Core

```
public async Task<RecipeDetailViewModel> GetRecipeDetail(int id)      #A
{
    return await _context.Recipes                                     #B
        .Where(x => x.RecipeId == id)                           #C
        .Select(x => new RecipeDetailViewModel           #D
        {                                                #D
            Id = x.RecipeId,                             #D
            Name = x.Name,                               #D
            Method = x.Method,                          #D
            Ingredients = x.Ingredients                        #E
            .Select(item => new RecipeDetailViewModel.Item     #E
            {                                                  #E
                Name = item.Name,                              #E
                Quantity = $"{item.Quantity} {item.Unit}"      #E
            })                                                #E
        })
        .SingleOrDefaultAsync();       #F
}
```

#A The id of the recipe to load is passed as a parameter.
#B As before, a query starts from a DbSet property.
#C Limit the query to the recipe with the provided id.
#D Map the Recipe to a RecipeDetailViewModel.
#E Load and map linked Ingredients as part of the same query.
#F Execute the query and map the data to the view model.

Notice that, as well as mapping the `Recipe` to a `RecipeDetailViewModel`, you also map the related `Ingredient`s for a `Recipe`, as though you're working with the objects directly in memory. This is one of the advantages of using an ORM—you can easily map child objects and let EF Core decide how best to build the underlying queries to fetch the data.

> **NOTE** EF Core logs all the SQL statements it runs as `LogLevel.Information` events by default, so you can easily see what queries are being run against the database.

Our app is definitely shaping up; you can create new recipes, view them all in a list, and drill down to view individual recipes with their ingredients and method. Pretty soon though, someone's going to introduce a typo and want to change their data. To do this, you'll have to implement the *U* in CRUD: update.

12.4.4 Updating a model with changes

Updating entities when they have changed is generally the hardest part of CRUD operations, as there are so many variables. Figure 12.14 gives an overview of this process as it applies to your recipe app.

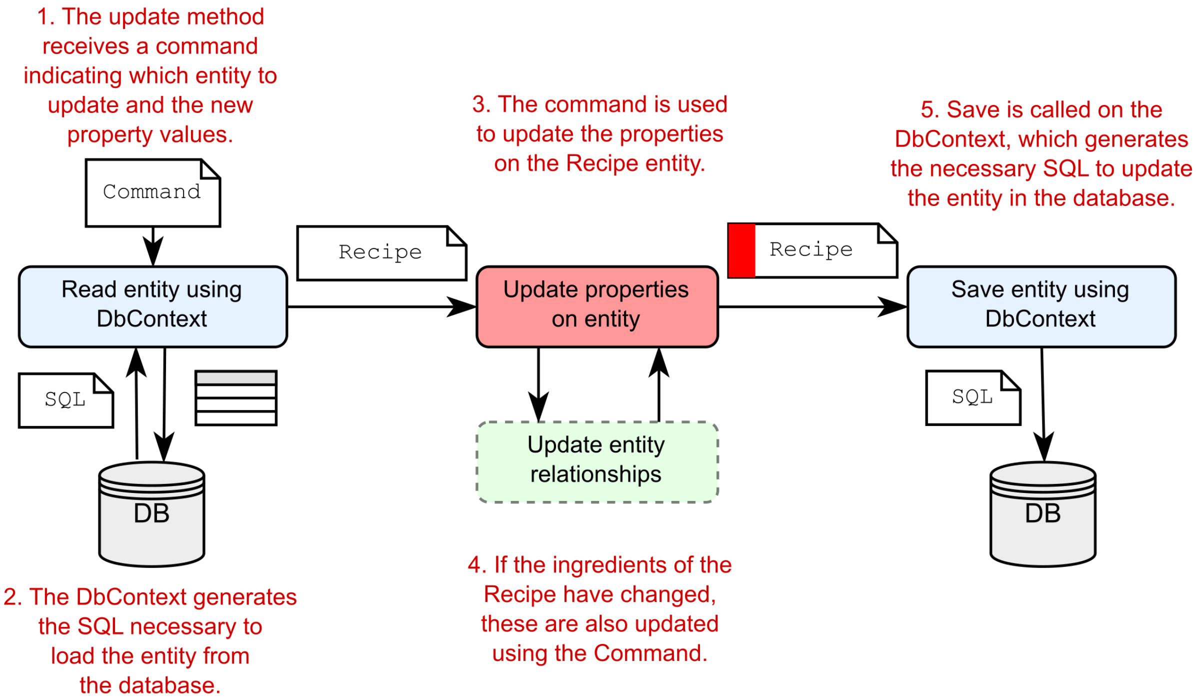

1. The update method receives a command indicating which entity to update and the new property values.
Command
Read entity using DbContext
SQL
DB
2. The DbContext generates the SQL necessary to load the entity from the database.
Recipe
3. The command is used to update the properties on the Recipe entity.
Update properties on entity
Update entity relationships
4. If the ingredients of the Recipe have changed, these are also updated using the Command.
Recipe
5. Save is called on the DbContext, which generates the necessary SQL to update the entity in the database.
Save entity using DbContext
SQL
DB

Figure 12.14 Updating an entity involves three steps: read the entity using EF Core, update the properties of the entity, and call `SaveChangesAsync()` on the `DbContext` to generate the SQL to update the correct rows in the database.

I’m not going to handle the relationship aspect in this book because that’s generally a complex problem, and how you tackle it depends on the specifics of your data model. Instead, I’ll focus on updating properties on the `Recipe` entity itself.[58]

For web applications, when you update an entity, you’ll typically follow the steps outlined in Figure 12.14:

1. Read the entity from the database
2. Modify the entity’s properties
3. Save the changes to the database

You’ll encapsulate these three steps in a method on `RecipeService`, `UpdateRecipe`, which takes `UpdateRecipeCommand`, containing the changes to make to the `Recipe` entity.

[58] For a detailed discussion on handling relationship updates in EF Core, see *EF Core in Action* by Jon P Smith (Manning, 2018) https://livebook.manning.com#!/book/smith3/Chapter-3/109.

NOTE As with the `Create` command, you don't directly modify the entities in the Razor Page, ensuring you keep the UI concern separate from the business logic.

The following listing shows the `RecipeService.UpdateRecipe` method, which updates the `Recipe` entity. It's the three steps we defined previously to read, modify, and save the entity. I've extracted the code to update the recipe with the new values to a helper method.

Listing 12.9 Updating an existing entity with EF Core

```
public async Task UpdateRecipe(UpdateRecipeCommand cmd)
{
    var recipe = await _context.Recipes.FindAsync(cmd.Id);        #A
    if(recipe == null) {                                        #B
        throw new Exception("Unable to find the recipe");       #B
    }                                                           #B
    UpdateRecipe(recipe, cmd);                                  #C
    await _context.SaveChangesAsync();                           #D
}

static void UpdateRecipe(Recipe recipe, UpdateRecipeCommand cmd)    #E
{                                                                   #E
    recipe.Name = cmd.Name;                                         #E
    recipe.TimeToCook =                                             #E
        new TimeSpan(cmd.TimeToCookHrs, cmd.TimeToCookMins, 0);     #E
    recipe.Method = cmd.Method;                                     #E
    recipe.IsVegetarian = cmd.IsVegetarian;                         #E
    recipe.IsVegan = cmd.IsVegan;                                   #E
}                                                                   #E
```

#A Find is exposed directly by Recipes and simplifies reading an entity by id.
#B If an invalid id is provided, recipe will be null.
#C Set the new values on the Recipe entity.
#D Execute the SQL to save the changes to the database.
#E A helper method for setting the new properties on the Recipe entity

In this example, I read the `Recipe` entity using the `FindAsync(id)` method exposed by `DbSet`. This is a simple helper method for loading an entity by its ID, in this case `RecipeId`. I could've written a similar query using LINQ as

```
_context.Recipes.Where(r=>r.RecipeId == cmd.Id).FirstOrDefault();
```

Using `FindAsync()` or `Find()` is a little more declarative and concise.

TIP `Find` is actually a bit more complicated. `Find` first checks to see if the entity is already being tracked in EF Core's `DbContext`. If it is (because the entity was previously loaded this request) then the entity is returned immediately without calling the DB. This can obviously be faster if the entity *is* tracked, but it can also be slower if you *know* the entity *isn't* being tracked yet.

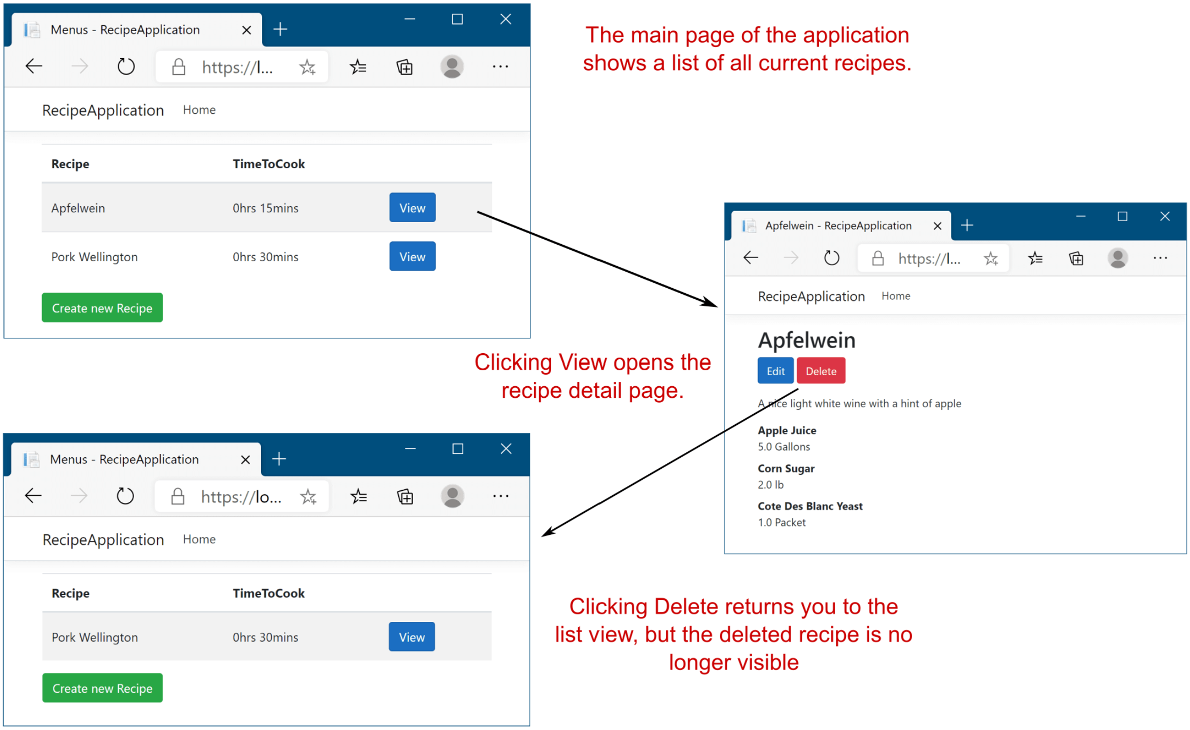
You may be wondering how EF Core knows which columns to update when you call `SaveChangesAsync()`. The simplest approach would be to update every column—if the field hasn't changed, then it doesn't matter if you write the same value again. But EF Core is a bit more clever than that.

EF Core internally tracks the *state* of any entities it loads from the database. It creates a snapshot of all the entity's property values, so it can track which ones have changed. When you call `SaveChanges()`, EF Core compares the state of any tracked entities (in this case, the `Recipe` entity) with the tracking snapshot. Any properties that have been changed are included in the `UPDATE` statement sent to the database, unchanged properties are ignored.

> **NOTE** EF Core provides other mechanisms to track changes, as well as options to disable change-tracking altogether. See the documentation or Jon P Smith's *EF Core in Action, Second Edition* (Manning, 2021) for details https://livebook.manning.com/book/entity-framework-core-in-action-second-edition/chapter-3.

With the ability to update recipes, you're almost done with your recipe app. "But wait," I hear you cry, "we haven't handled the *D* in CRUD—delete!" And that's true, but in reality, I've found few occasions when you *want* to delete data.

Let's consider the requirements for deleting a recipe from the application, as shown in figure 12.15. You need to add a (scary-looking) Delete button next to a recipe. After the user clicks Delete, the recipe is no longer visible in the list and can't be viewed.



Menus - RecipeApplication

RecipeApplication Home

Recipe	TimeToCook	
Apfelwein	0hrs 15mins	View
Pork Wellington	0hrs 30mins	View

Create new Recipe

The main page of the application shows a list of all current recipes.

Clicking View opens the recipe detail page.

Apfelwein - RecipeApplication

RecipeApplication Home

Apfelwein

Edit Delete

A nice light white wine with a hint of apple

Apple Juice
5.0 Gallons

Corn Sugar
2.0 lb

Cote Des Blanc Yeast
1.0 Packet

Menus - RecipeApplication

RecipeApplication Home

Recipe	TimeToCook	
Pork Wellington	0hrs 30mins	View

Create new Recipe

Clicking Delete returns you to the list view, but the deleted recipe is no longer visible

Figure 12.15 The desired behavior when deleting a recipe from the app. Clicking Delete should return you to the application's main list view, with the deleted recipe no longer visible.

Now, you *could* achieve this by deleting the recipe from the database, but the problem with data is once it's gone, it's gone! What if a user accidentally deletes a record? Also, deleting a row from a relational database typically has implications on other entities. For example, you can't delete a row from the Recipe table in your application without also deleting all the Ingredient rows that reference it, thanks to the foreign key constraint on `Ingredient.RecipeId`.

EF Core can easily handle these *true deletion* scenarios for you with the `DbContext.Remove(entity)` command but, typically, what you *mean* when you find a need to delete data is to "archive" it or hide it from the UI. A common approach to handling this scenario is to include some sort of "Is this entity deleted" flag on your entity, such as the `IsDeleted` flag I included on the `Recipe` entity:

```
public bool IsDeleted {get;set;}
```

If you take this approach then, suddenly, deleting data becomes simpler, as it's nothing more than an update to the entity. No more issues of lost data and no more referential integrity problems.

NOTE The main exception I've found to this pattern is when you're storing your users' personally identifying information. In these cases, you may be duty-bound (and, potentially, legally bound) to scrub their information from your database on request.

With this approach, you can create a *delete* method on `RecipeService`, which updates the `IsDeleted` flag, as shown in the following listing. In addition, you should ensure you have `Where()` clauses in all the other methods in your `RecipeService`, to ensure you can't display a deleted `Recipe`, as you saw in listing 12.9, for the `GetRecipes()` method.

Listing 12.10 Marking entities as deleted in EF Core

```
public async Task DeleteRecipe(int recipeId)
{
    var recipe = await _context.Recipes.FindAsync(recipeId);       #A
    if(recipe is null) {                                     #B
        throw new Exception("Unable to find the recipe");    #B
    }                                                        #B
    recipe.IsDeleted = true;      #C
    await _context.SaveChangesAsync();       #D
}
```

#A Fetch the Recipe entity by id.
#B If an invalid id is provided, recipe will be null.
#C Mark the Recipe as deleted.
#D Execute the SQL to save the changes to the database.

This approach satisfies the requirements—it removes the recipe from the UI of the application—but it simplifies a number of things. This *soft delete* approach won't work for all scenarios, but I've found it to be a common pattern in projects I've worked on.

TIP EF Core has a handy feature called *global query filters*. These allow you to specify a `Where` clause at the model level, so you could, for example, ensure that EF Core *never* loads `Recipes` for which `IsDeleted` is `true`. This is also useful for segregating data in a multi-tenant environment. See the documentation for details: https://docs.microsoft.com/ef/core/querying/filters.

We're almost at the end of this chapter on EF Core. We've covered the basics of adding EF Core to your project and using it to simplify data access, but you'll likely need to read more into EF Core as your apps become more complex. In the final section of this chapter, I'd like to pinpoint a number of things you need to take into consideration before using EF Core in your own applications, so you're familiar with some of the issues you'll face as your apps grow.

12.5 Using EF Core in production applications

This book is about ASP.NET Core, not EF Core, so I didn't want to spend too much time exploring EF Core. This chapter should've given you enough to get up and running, but you'll definitely need to learn more before you even think about putting EF Core into production. As I've said several times, I recommend *EF Core in Action* by Jon P Smith (Manning, 2018) for details (https://livebook.manning.com/#!/book/smith3/Chapter-11/), or exploring the EF Core documentation site at https://docs.microsoft.com/ef/core/.

The following topics aren't essential to getting started with EF Core, but you'll quickly come up against them if you build a production-ready app. This section isn't a prescriptive guide to how to tackle each of these; it's more a set of things to consider before you dive into production!

- *Scaffolding of columns*—EF Core uses conservative values for things like `string` columns by allowing strings of large or unlimited length. In practice, you may want to restrict these and other data types to sensible values.
- *Validation*—You can decorate your entities with `DataAnnotations` validation attributes, but EF Core won't automatically validate the values before saving to the database. This differs from EF 6.x behavior, in which validation was automatic.
- *Handling concurrency*—EF Core provides a few ways to handle concurrency, where multiple users attempt to update an entity at the same time. One partial solution is by using `Timestamp` columns on your entities.
- *Synchronous vs. asynchronous*—EF Core provides both synchronous and asynchronous commands for interacting with the database. Often, async is better for web apps, but there are nuances to this argument that make it impossible to recommend one approach over the other in all situations.

EF Core is a great tool for being productive when writing data-access code, but there are some aspects of working with a database that are unavoidably awkward. The issue of database management is one of the thorniest issues to tackle. This book is about ASP.NET Core, not EF Core, so I don't want to dwell on database management too much. Having said that, most web applications use some sort of database, so the following issues are likely to impact you at some point.

- *Automatic migrations*—If you automatically deploy your app to production as part of some sort of DevOps pipeline, you'll inevitably need some way of applying migrations to a database automatically. You can tackle this in several ways, such as scripting the .NET Core tool, applying migrations in your app's startup code, or using a custom tool. Each approach has its pros and cons.
- *Multiple web hosts*—One specific consideration is whether you have multiple web servers hosting your app, all pointing to the same database. If so, then applying migrations in your app's startup code becomes harder, as you must ensure only one app can migrate the database at a time.
- *Making backward-compatible schema changes*—A corollary of the multiple-web host approach is that you'll often be in a situation where your app is accessing a database that has a *newer* schema than the app thinks. That means you should normally endeavor to make schema changes backward-compatible wherever possible.
- *Storing migrations in a different assembly*—In this chapter, I included all my logic in a single project, but in larger apps, data access is often in a different project to the web

app. For apps with this structure, you must use slightly different commands when using the .NET CLI or PowerShell cmdlets.

- *Seeding data*—When you first create a database, you often want it to have some initial *seed* data, such as a default user. EF 6.X had a mechanism for seeding data built in, whereas EF Core requires you to explicitly seed your database yourself.

How you choose to handle each of these issues will depend on the infrastructure and deployment approach you take with your application. None of them are particularly fun to tackle but they're an unfortunate necessity. Take heart though, they can all be solved one way or another!

That brings us to the end of this chapter on EF Core. In the next chapter, we'll look at one of the slightly more advanced topics of MVC and Razor Pages, the filter pipeline, and how you can use it to reduce duplication in your code.

12.6 Summary

- EF Core is an object-relational mapper (ORM) that lets you interact with a database by manipulating standard POCO classes, called entities, in your application. This can reduce the amount of SQL and database knowledge you need to have to be productive.
- EF Core maps entity classes to tables, properties on the entity to columns in the tables, and instances of entity objects to rows in these tables. Even if you use EF Core to avoid working with a database directly, you need to keep this mapping in mind.
- EF Core uses a database-provider model that lets you change the underlying database without changing any of your object manipulation code. EF Core has database providers for Microsoft SQL Server, SQLite, PostgreSQL, MySQL, and many others.
- EF Core is cross-platform and has good performance for an ORM, but has a different feature set to EF 6.x. Nevertheless, EF Core is recommended for all new applications over EF 6.x.
- EF Core stores an internal representation of the entities in your application and how they map to the database, based on the `DbSet<T>` properties on your application's `DbContext`. EF Core builds a model based on the entity classes themselves and any other entities they reference.
- You add EF Core to your app by adding a NuGet database provider package. You should also install the Design packages for EF Core. This works in conjunction with the .NET Core tools to generate and apply migrations to a database.
- EF Core includes many conventions for how entities are defined, such as primary keys and foreign keys. You can customize how entities are defined either declaratively, using `DataAnotations`, or using a fluent API.
- Your application uses a `DbContext` to interact with EF Core and the database. You register it with a DI container using `AddDbContext<T>`, defining the database provider and providing a connection string. This makes your `DbContext` available in the DI container throughout your app.

- EF Core uses migrations to track changes to your entity definitions. They're used to ensure your entity definitions, EF Core's internal model, and the database schema all match.
- After changing an entity, you can create a migration either using the .NET Core tool or using Visual Studio PowerShell cmdlets.
- To create a new migration with the .NET CLI, run `dotnet ef migrations add NAME` in your project folder, where `NAME` is the name you want to give the migration. This compares your current `DbContext` snapshot to the previous version and generates the necessary SQL statements to update your database.
- You can apply the migration to the database using `dotnet ef database update`. This will create the database if it doesn't already exist and apply any outstanding migrations.
- EF Core doesn't interact with the database when it creates migrations, only when you explicitly update the database, so you can still create migrations when you're offline.
- You can add entities to an EF Core database by creating a new entity, `e`, calling `_context.Add(e)` on an instance of your application's data context, `_context`, and calling `_context.SaveChangesAsync()`. This generates the necessary SQL `INSERT` statements to add the new rows to the database.
- You can load records from a database using the `DbSet<T>` properties on your app's `DbContext`. These expose the `IQueryable` interface, so you can use LINQ statements to filter and transform the data in the database before it's returned.
- Updating an entity consists of three steps: read the entity from the database, modify the entity, and save the changes to the database. EF Core will keep track of which properties have changed so that it can optimize the SQL it generates.
- You can delete entities in EF Core using the `Remove` method, but you should consider carefully whether you need this functionality. Often a *soft delete* technique using an `IsDeleted` flag on entities is safer and easier to implement.
- This chapter only covers a subset of the issues you must consider when using EF Core in your application. Before using it in a production app, you should consider, among other things: the data types generated for fields, validation, how to handle concurrency, the seeding of initial data, handling migrations on a running application, and handling migrations in a web-farm scenario.

13

The MVC and Razor Pages filter pipeline

This chapter covers

- **The filter pipeline and how it differs from middleware**
- **Creating custom filters to refactor complex action methods**
- **Using authorization filters to protect your action methods and Razor Pages**
- **Short-circuiting the filter pipeline to bypass action and page handler execution**
- **Injecting dependencies into filters**

In part 1, I covered the MVC and Razor Pages frameworks of ASP.NET Core in some detail. You learned how routing is used to select an action method or Razor Page to execute. You also saw model binding, validation, and how to generate a response by returning an `IActionResult` from your actions and page handlers. In this chapter, I'm going to head deeper into the MVC/Razor Pages frameworks and look at the *filter pipeline*, sometimes called the *action invocation pipeline*.

MVC and Razor Pages use several built-in filters to handle crosscutting concerns, such as authorization (controlling which users can access which action methods and pages in your application). Any application that has the concept of users will use authorization filters as a minimum, but filters are much more powerful than this single use case.

This chapter describes the filter pipeline primarily in the context of an API controller request. You'll learn how to create custom filters that you can use in your own apps, and how you can use them to reduce duplicate code in your action methods. You'll learn how to customize your application's behavior for specific actions, as well as how to apply filters globally to modify all of the actions in your app.

You'll also learn how the filter pipeline applies to Razor Pages. The Razor Pages filter pipeline is almost identical to the MVC/API controller filter pipeline, so we'll focus on where it differs. You'll see how to use page filters in your Razor Pages and learn how they differ from action filters.

Think of the filter pipeline as a mini middleware pipeline running inside the MVC and Razor Pages frameworks. Like the middleware pipeline in ASP.NET Core, the filter pipeline consists of a series of components connected as a pipe, so the output of one filter feeds into the input of the next.

This chapter starts by looking at the similarities and differences between filters and middleware, and when you should choose one over the other. You'll learn about all the different types of filters and how they combine to create the filter pipeline for a request that reaches the MVC or Razor Pages framework.

In section 13.2, I'll take you through each filter type in detail, how they fit into the MVC pipeline, and what to use them for. For each one, I'll provide example implementations that you might use in your own application.

A key feature of filters is the ability to short-circuit a request by generating a response and halting progression through the filter pipeline. This is similar to the way short-circuiting works in middleware, but there are subtle differences. On top of that, the exact behavior is slightly different for each filter, which I cover in section 13.3.

You typically add filters to the pipeline by implementing them as attributes added to your controller classes, action methods, and Razor Pages. Unfortunately, you can't easily use DI with attributes due to the limitations of C#. In section 13.4, I'll show you how to use the `ServiceFilterAttribute` and `TypeFilterAttribute` base classes to enable dependency injection in your filters.

Before we can start writing code, we should get to grips with the basics of the filter pipeline. The first section of this chapter explains what the pipeline is, why you might want to use it, and how it differs from the middleware pipeline.

13.1 Understanding filters and when to use them

In this section you'll learn all about the filter pipeline. You'll see where it fits in the lifecycle of a typical request, how it differs between MVC and Razor Pages, and how filters differ from middleware. You'll learn about the six types of filter, how you can add them to your own apps, and how to control the order in which they execute when handling a request.

The filter pipeline is a relatively simple concept, in that it provides *hooks* into the normal MVC request, as shown in figure 13.1. For example, say you wanted to ensure that users can create or edit products on an e-commerce app *only* if they're logged in. The app would redirect anonymous users to a login page instead of executing the action.

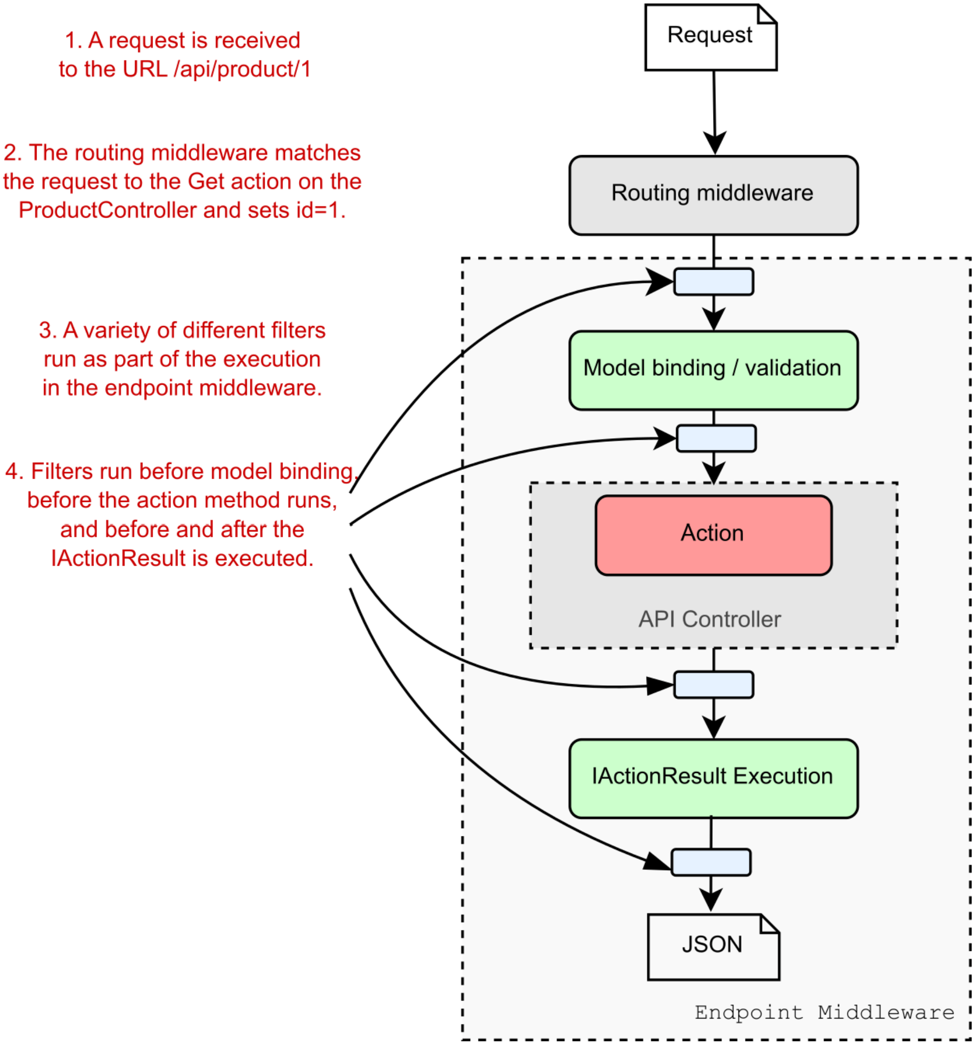

1. A request is received to the URL /api/product/1
2. The routing middleware matches the request to the Get action on the ProductController and sets id=1.
3. A variety of different filters run as part of the execution in the endpoint middleware.
4. Filters run before model binding, before the action method runs, and before and after the IActionResult is executed.
Request
Routing middleware
Model binding / validation
Action
API Controller
IActionResult Execution
JSON
Endpoint Middleware

Figure 13.1 Filters run at multiple points in the `EndpointMiddleware` as part of the normal handling of an MVC request. A similar pipeline exists for Razor Page requests.

Without filters, you'd need to include the same code to check for a logged-in user at the start of each specific action method. With this approach the MVC framework would still execute the model binding and validation, even if the user were not logged in.

With filters, you can use the *hooks* in the MVC request to run common code across all, or a subset of, requests. This way, you can do a wide range of things, such as

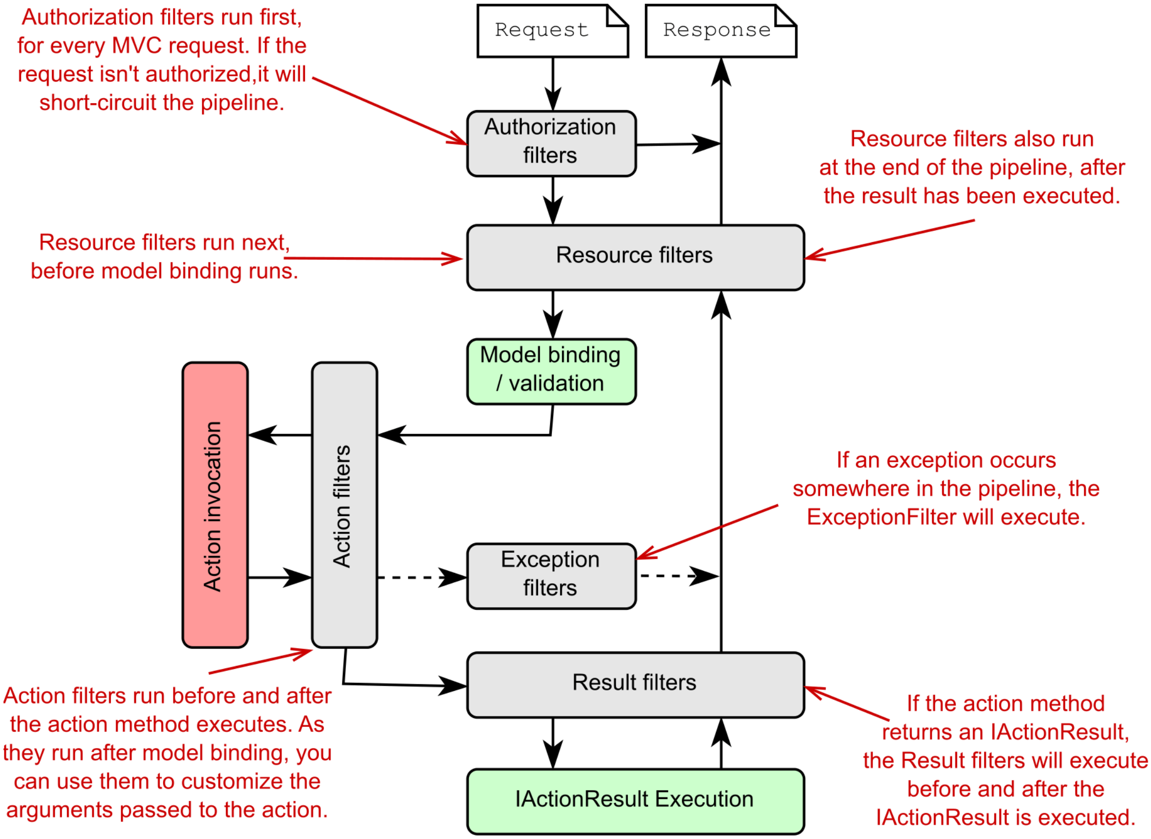
- Ensuring a user is logged in before an action method, model binding, or validation runs

- Customizing the output format of particular action methods
- Handling model validation failures before an action method is invoked
- Catching exceptions from an action method and handling them in a special way

In many ways, the filter pipeline is like a middleware pipeline, but restricted to MVC and Razor Pages requests only. Like middleware, filters are good for handling crosscutting concerns for your application and are a useful tool for reducing code duplication in many cases.

13.1.1 The MVC filter pipeline

As you saw in figure 13.1, filters run at a number of different points in an MVC request. This linear view of an MVC request and the filter pipeline that I've used so far doesn't *quite* match up with how these filters execute. There are five types of filter that apply to MVC requests, each of which runs at a different *stage* in the MVC framework, as shown in figure 13.2.



Authorization filters run first, for every MVC request. If the request isn't authorized,it will short-circuit the pipeline.

Resource filters also run at the end of the pipeline, after the result has been executed.

Resource filters run next, before model binding runs.

If an exception occurs somewhere in the pipeline, the ExceptionFilter will execute.

Action filters run before and after the action method executes. As they run after model binding, you can use them to customize the arguments passed to the action.

If the action method returns an IActionResult, the Result filters will execute before and after the IActionResult is executed.

Figure 13.2 The MVC filter pipeline, including the five different filter stages. Some filter stages (resource, action, and result) run twice, before and after the remainder of the pipeline.

Each filter stage lends itself to a particular use case, thanks to its specific location in the pipeline, with respect to model binding, action execution, and result execution.

- *Authorization filters*—These run first in the pipeline, so are useful for protecting your APIs and action methods. If an authorization filter deems the request unauthorized, it will short-circuit the request, preventing the rest of the filter pipeline (or action) from running.
- *Resource filters*—After authorization, resource filters are the next filters to run in the pipeline. They can also execute at the *end* of the pipeline, in much the same way that middleware components can handle both the incoming request and the outgoing response. Alternatively, resource filters can completely short-circuit the request pipeline and return a response directly.

  Thanks to their early position in the pipeline, resource filters can have a variety of uses. You could add metrics to an action method, prevent an action method from executing if an unsupported content type is requested, or, as they run before model binding, control the way model binding works for that request.
- *Action filters*—Action filters run just before and after an action method is executed. As model binding has already happened, action filters let you manipulate the arguments to the method—before it executes—or they can short-circuit the action completely and return a different `IActionResult`. Because they also run after the action executes, they can optionally customize an `IActionResult` returned by the action before the action result is executed.
- *Exception filters*—Exception filters can catch exceptions that occur in the filter pipeline and handle them appropriately. You can use exception filters to write custom MVC-specific error-handling code, which can be useful in some situations. For example, you could catch exceptions in API actions and format them differently from exceptions in your Razor Pages.
- *Result filters*—Result filters run before and after an action method's `IActionResult` is executed. You can use result filters to control the execution of the result, or even to short-circuit the execution of the result.

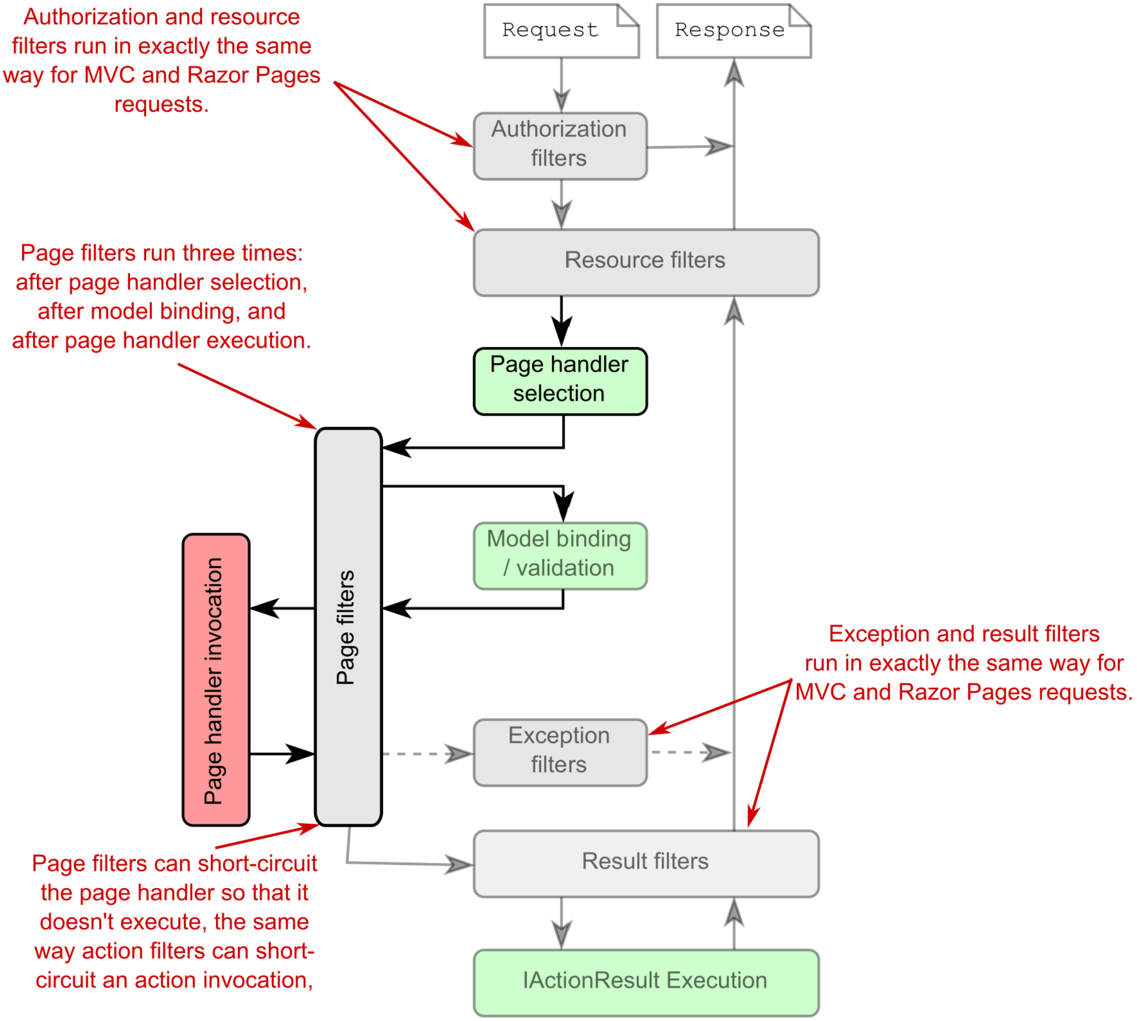
Exactly which filter you pick to implement will depend on the functionality you're trying to introduce. Want to short-circuit a request as early as possible? Resource filters are a good fit. Need access to the action method parameters? Use an action filter.

Think of the filter pipeline as a small middleware pipeline that lives by itself in the MVC framework. Alternatively, you could think of filters as *hooks* into the MVC action invocation process, which let you run code at a particular point in a request's lifecycle.

The example above describes how the filter pipeline works for MVC controllers, such as you would use to create APIs, but Razor Pages uses an almost identical filter pipeline.

13.1.2 The Razor Pages Filter pipeline

The Razor Pages framework uses the same underlying architecture as API controllers, so it’s perhaps not surprising that the filter pipeline is virtually identical. The only difference between the pipelines is that Razor Pages do not use action filters. Instead, they use page filters, as shown in figure 13.3.



Authorization and resource filters run in exactly the same way for MVC and Razor Pages requests.

Page filters run three times: after page handler selection, after model binding, and after page handler execution.

Exception and result filters run in exactly the same way for MVC and Razor Pages requests.

Page filters can short-circuit the page handler so that it doesn't execute, the same way action filters can short-circuit an action invocation,

Figure 13.3 The Razor Pages filter pipeline, including the five different filter stages. Authorization, resource, exception, and result filters execute in exactly the same way as for the MVC pipeline. Page filters are specific to Razor Pages and execute in three places: after page hander selection, after model binding and validation, and after page handler execution.

The authorization, resource, exception, and result filters are exactly the same filters as you saw for the MVC pipeline. They execute in the same way, serve the same purposes, and can be short-circuited in the same way.

> **NOTE** These filters are literally the same classes shared between the Razor Pages and MVC frameworks. For example, if you create an exception filter and register it globally, the filter will apply to all your API controllers and all your Razor Pages equally.

The difference with the Razor Pages filter pipeline is that it uses *page filters* instead of action filters. In contrast to other filter types, page filters run three times in the filter pipeline:

- *After page handler selection*—After the resource filters have executed, a page handler is selected, based on the request's HTTP verb and the `{handler}` route value, as you learned in chapter 5. After page handler selection, a page filter method executes for the first time. You can't short-circuit the pipeline at this stage, and model binding and validation has not yet executed.
- *After model binding*—After the first page filter execution, the request is model bound to the Razor Page's binding models and is validated. This execution is highly analogous to the action filter execution for API controllers. At this point you could manipulate the model-bound data or short-circuit the page handler execution completely by returning a different `IActionResult`.
- *After page handler execution*—If you don't short circuit the page handler execution, the page filter runs a third and final time after the page handler has executed. At this point you could customize the `IActionResult` returned by the page handler before the result is executed.

The triple-execution of page filters makes it a bit harder to visualize the pipeline, but you can generally just think of them as beefed-up action filters. Everything you can do with an action filter you can do with a page filter. Plus, you can hook-in after page handler selection if necessary.

> **TIP** Each execution of a filter executes a different method of the appropriate interface, so it's easy to know where you are in the pipeline, and to only execute a filter in one of its possible locations if you wish.

One of the main questions I hear when people learn about filters in ASP.NET Core is "Why do we need them?" If the filter pipeline is like a mini middleware pipeline, why not use a middleware component directly, instead of introducing the filter concept? That's an excellent point, which I'll tackle in the next section.

13.1.3 Filters or middleware: which should you choose?

The filter pipeline is similar to the middleware pipeline in many ways, but there are several subtle differences that you should consider when deciding which approach to use. When considering the similarities, they have three main parallels:

- *Requests pass through a middleware component on the way "in" and responses pass through again on the way "out."* Resource, action, and result filters are also two-way, though authorization and exception filters run only once for a request, and page filters run three times!
- *Middleware can short-circuit a request by returning a response, instead of passing it on to later middleware.* Filters can also short-circuit the filter pipeline by returning a response.
- *Middleware is often used for crosscutting application concerns, such as logging, performance profiling, and exception handling.* Filters also lend themselves to crosscutting concerns.

In contrast, there are three main differences between middleware and filters:

- Middleware can run for all requests; filters will only run for requests that reach the `EndpointMiddleware` and execute an API controller action or Razor Page.
- Filters have access to MVC constructs such as `ModelState` and `IActionResults`. Middleware, in general, is independent from MVC and Razor Pages, so can't use these concepts.
- Filters can be easily applied to a subset of requests; for example, all actions on a single controller, or a single Razor Page. Middleware doesn't have this concept as a first-class idea (though you could achieve something similar with custom middleware components).

That's all well and good, but how should we interpret these differences? When should we choose one over the other?

I like to think of middleware versus filters as a question of specificity. Middleware is the more general concept, so has the wider reach. If the functionality you need has no MVC-specific requirements, then you should use a middleware component. Exception handling is a great example of this; exceptions could happen anywhere in your application, and you need to handle them, so using exception-handling middleware makes sense.

On the other hand, if you *do* need access to MVC constructs, or you want to behave differently for some MVC actions, then you should consider using a filter. Ironically, this can also be applied to exception handling. You don't want exceptions in your Web API controllers to automatically generate HTML error pages when the client is expecting JSON. Instead, you could use an exception filter on your Web API actions to render the exception to JSON, while letting the exception-handling middleware catch errors from Razor Pages in your app.

> **TIP** Where possible, consider using middleware for crosscutting concerns. Use filters when you need different behavior for different action methods, or where the functionality relies on MVC concepts like `ModelState` validation.

The middleware versus filters argument is a subtle one, and it doesn't matter which you choose as long as it works for you. You can even use middleware components *inside* the filter pipeline as filters, but that's outside the scope of this book.

> **TIP** The middleware as filters feature was introduced in ASP.NET Core 1.1 and is also available in later versions. The canonical use case is for localizing requests to multiple languages. I have a blog series on how to use the feature here: https://andrewlock.net/series/adding-a-url-culture-provider-using-middleware-as-filters/.

Filters can be a little abstract in isolation, so in the next section, we'll look at some code and learn how to write a custom filter in ASP.NET Core.

13.1.4 Creating a simple filter

In this section, I show how to create your first filters; in section 13.1.5, you'll see how to apply them to MVC controllers and actions. We'll start small, creating filters that just write to the console, but in section 13.2, we'll look at some more practical examples and discuss some of their nuances.

You implement a filter for a given stage by implementing one of a pair of interfaces—one synchronous (sync), one asynchronous (async):

- *Authorization filters*—`IAuthorizationFilter` or `IAsyncAuthorizationFilter`
- *Resource filters*—`IResourceFilter` or `IAsyncResourceFilter`
- *Action filters*—`IActionFilter` or `IAsyncActionFilter`
- *Page filters*—`IPageFilter` or `IAsyncPageFilter`
- *Exception filters*—`IExceptionFilter` or `IAsyncExceptionFilter`
- *Result filters*—`IResultFilter` or `IAsyncResultFilter`

You can use any POCO class to implement a filter, but you'll typically implement them as C# attributes, which you can use to decorate your controllers, actions, and Razor Pages, as you'll see in section 13.1.5. You can achieve the same results with either the sync or async interface, so which you choose should depend on whether any services you call in the filter require async support.

> **NOTE** You should implement *either* the sync interface *or* the async interface, *not both*. If you implement both, then only the async interface will be used.

Listing 13.1 shows a resource filter that implements `IResourceFilter` and writes to the console when it executes. The `OnResourceExecuting` method is called when a request first reaches the resource filter stage of the filter pipeline. In contrast, the `OnResourceExecuted` method is called after the rest of the pipeline has executed; after model binding, action execution, result execution, and all intermediate filters have run.

Listing 13.1 Example resource filter implementing `IResourceFilter`

```
public class LogResourceFilter : Attribute, IResourceFilter
```

```
{
    public void OnResourceExecuting(         #A
        ResourceExecutingContext context)    #B
    {
        Console.WriteLine("Executing!");
    }

    public void OnResourceExecuted(          #C
        ResourceExecutedContext context)     #D
    {
        Console.WriteLine("Executed”");
    }
}
```

#A Executed at the start of the pipeline, after authorization filters.
#B The context contains the HttpContext, routing details, and information about the current action.
#C Executed after model binding, action execution, and result execution.
#D Contains additional context information, such as the IActionResult returned by the action.

The interface methods are simple and are similar for each stage in the filter pipeline, passing a context object as a method parameter. Each of the two-method sync filters has an `*Executing` and an `*Executed` method. The type of the argument is different for each filter, but it contains all the details for the filter pipeline.

For example, the `ResourceExecutingContext` passed to the resource filter contains the `HttpContext` object itself, details about the route that selected this action, details about the action itself, and so on. Contexts for later filters will contain additional details, such as the action method arguments for an action filter and the `ModelState`.

The context object for the `ResourceExecutedContext` method is similar, but it also contains details about how the rest of the pipeline executed. You can check whether an unhandled exception occurred, you can see if another filter from the same stage short-circuited the pipeline, or you can see the `IActionResult` used to generate the response.

These context objects are powerful and are the key to advanced filter behaviors like short-circuiting the pipeline and handling exceptions. We’ll make use of them in section 13.2 when creating more complex filter examples.

The async version of the resource filter requires implementing a single method, as shown in listing 13.2. As for the sync version, you’re passed a `ResourceExecutingContext` object as an argument, and you’re passed a delegate representing the remainder of the filter pipeline. You must call this delegate (asynchronously) to execute the remainder of the pipeline, which will return an instance of `ResourceExecutedContext`.

Listing 13.2 Example resource filter implementing `IAsyncResourceFilter`

```
public class LogAsyncResourceFilter : Attribute, IAsyncResourceFilter
{
    public async Task OnResourceExecutionAsync(                 #A
        ResourceExecutingContext context,
        ResourceExecutionDelegate next)                         #B
    {
        Console.WriteLine("Executing async!");                        #C
        ResourceExecutedContext executedContext = await next();     #D
        Console.WriteLine("Executed async!");                       #E
    }
}
```

#A Executed at the start of the pipeline, after authorization filters.
#B You’re provided a delegate, which encapsulates the remainder of the filter pipeline.
#C Called before the rest of the pipeline executes.
#D Executes the rest of the pipeline and obtains a ResourceExecutedContext instance
#E Called after the rest of the pipeline executes.

The sync and async filter implementations have subtle differences, but for most purposes they’re identical. I recommend implementing the sync version if possible, and only falling back to the async version if you need to.

You’ve created a couple of filters now, so we should look at how to use them in the application. In the next section, we’ll tackle two specific issues: how to control which requests execute your new filters and how to control the order in which they execute.

13.1.5 Adding filters to your actions, controllers, Razor Pages, and globally

In section 13.1.2, I discussed the similarities and differences between middleware and filters. One of those differences is that filters can be scoped to specific actions or controllers, so that they only run for certain requests. Alternatively, you can apply a filter globally, so that it runs for every MVC action and Razor Page.

By adding filters in different ways, you can achieve several different results. Imagine you have a filter that forces you to log in to execute an action. How you add the filter to your app will significantly change your app’s behavior:

- *Apply the filter to a single action or Razor Page*—Anonymous users could browse the app as normal, but if they tried to access the protected action or Razor Page, they would be forced to log in.
- *Apply the filter to a controller*—Anonymous users could access actions from other controllers, but accessing any action on the protected controller would force them to log in.
- *Apply the filter globally*—Users couldn’t use the app without logging in. Any attempt to access an action or Razor Page would redirect the user to the login page.

> **NOTE** **ASP.NET Core comes with just such a filter out of the box,** `AuthorizeFilter`**. I’ll discuss this filter in section 13.2.1, and you’ll be seeing a lot more of it in chapter 15.**

As I described in the previous section, you normally create filters as attributes, and for good reason—it makes it easy for you to apply them to MVC controllers, actions, and Razor Pages. In this section, you’ll see how to apply `LogResourceFilter` from listing 13.1 to an action, a controller, a Razor Page, and globally. The level at which the filter applies is called its *scope*.

> **DEFINITION** The *scope* of a filter refers to how many different actions it applies to. A filter can be scoped to the action method, to the controller, to a Razor Page, or globally.

You'll start at the most specific scope—applying filters to a single action. The following listing shows an example of an MVC controller that has two action methods: one with `LogResourceFilter` and one without.

Listing 13.3 Applying filters to an action method

```
public class RecipeController : ControllerBase
{
    [LogResourceFilter]                 #A
    public IActionResult Index()        #A
    {                                   #A
        return Ok();                    #A
    }                                   #A
    public IActionResult View()         #B
    {                                   #B
        return OK();                    #B
    }                                   #B
}
```

#A LogResourceFilter will run as part of the pipeline when executing this action.
#B This action method has no filters at the action level.

Alternatively, if you want to apply the same filter to every action method, you could add the attribute at the controller scope, as in the next listing. Every action method in the controller will use `LogResourceFilter`, without having to specifically decorate each method:

Listing 13.4 Applying filters to a controller

```
[LogResourceFilter]                         #A
public class RecipeController : ControllerBase
{
    public IActionResult Index ()              #B
    {                                          #B
        return Ok();                           #B
    }                                          #B
    public IActionResult View()                #B
    {                                          #B
        return Ok();                           #B
    }                                          #B
}
```

#A The LogResourceFilter Is added to every action on the controller.
#B Every action in the controller is decorated with the filter.

For Razor Pages, you can apply attributes to your `PageModel`, as shown in the following listing. The filter applies to all page handlers in the Razor Page—it's not possible to apply filters to a single page handler, you must apply them at the page level.

Listing 13.5 Applying filters to a Razor Page

```
[LogResourceFilter]                      #A
public class IndexModel : PageModel
{
    public void OnGet()                  #B
    {                                      #B
    }                                      #B

    public void OnPost()                   #B
    {                                      #B
    }                                      #B
}
```

#A The LogResourceFilter Is added to the Razor Page's PageModel.
#B The filter applies to every page handler in the page.

Filters you apply as attributes to controllers, actions, and Razor Pages are automatically discovered by the framework when your application starts up. For common attributes, you can go one step further and apply filters globally, without having to decorate individual classes.

You add global filters in a different way to controller- or action-scoped filters—by adding a filter directly to the MVC services, when configuring your controllers and Razor Pages in `Startup`. This listing shows three equivalent ways to add a globally scoped filter.

Listing 13.6 Applying filters globally to an application

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers(options =>         #A
        {
            options.Filters.Add(new LogResourceFilter());       #B
            options.Filters.Add(typeof(LogResourceFilter));     #C
            options.Filters.Add<LogResourceFilter>();           #D
        });
    }
}
```

#A Adds filters using the MvcOptions object
#B You can pass an instance of the filter directly. . .
#C . . . or pass in the Type of the filter and let the framework create it.
#D Alternatively, the framework can create a global filter using a generic type parameter.

You can configure the `MvcOptions` by using the `AddControllers()` overload. When you configure filters globally, they apply both to controllers *and* to any Razor Pages in your application. If you're using Razor Pages in your application instead, there isn't an overload for configuring the `MvcOptions`. Instead you need to use the `AddMvcOptions()` extension method to configure the filters, as shown in the following listing.

Listing 13.7 Applying filters globally to a Razor Pages application

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages()                                #A
            .AddMvcOptions(options =>                           #B
            {
                options.Filters.Add(new LogResourceFilter());       #C
                options.Filters.Add(typeof(LogResourceFilter));     #C
                options.Filters.Add<LogResourceFilter>();           #C
            });
    }
}
```

#A This method doesn't let you pass a lambda to configure the MvcOptions
#B ...so you must use an extension method to add the filters to the MvcOptions object
#C You can configure the filters in any of the ways shown previously.

With potentially three different scopes in play, you'll often find action methods that have multiple filters applied to them, some applied directly to the action method, and others inherited from the controller or globally. The question then becomes: which filter runs first?

13.1.6 Understanding the order of filter execution

You've seen that the filter pipeline contains five different *stages*, one for each type of filter. These stages always run in the fixed order I described in sections 13.1.1 and 13.1.2. But within each stage, you can also have multiple filters of the same type (for example, multiple resource filters) that are part of a single action method's pipeline. These could all have multiple *scopes*, depending on how you added them, as you saw in the last section.

In this section, we're thinking about the *order of filters within a given stage* and how scope affects this. We'll start by looking at the default order, then move on to ways to customize the order to your own requirements.

THE DEFAULT SCOPE EXECUTION ORDER

When thinking about filter ordering, it's important to remember that resource, action, and result filters implement two methods: an `*Executing` before method and an `*Executed` after method. On top of that, page filters implement three methods! The order in which each method executes depends on the scope of the filter, as shown in figure 13.4 for the resource filter stage.

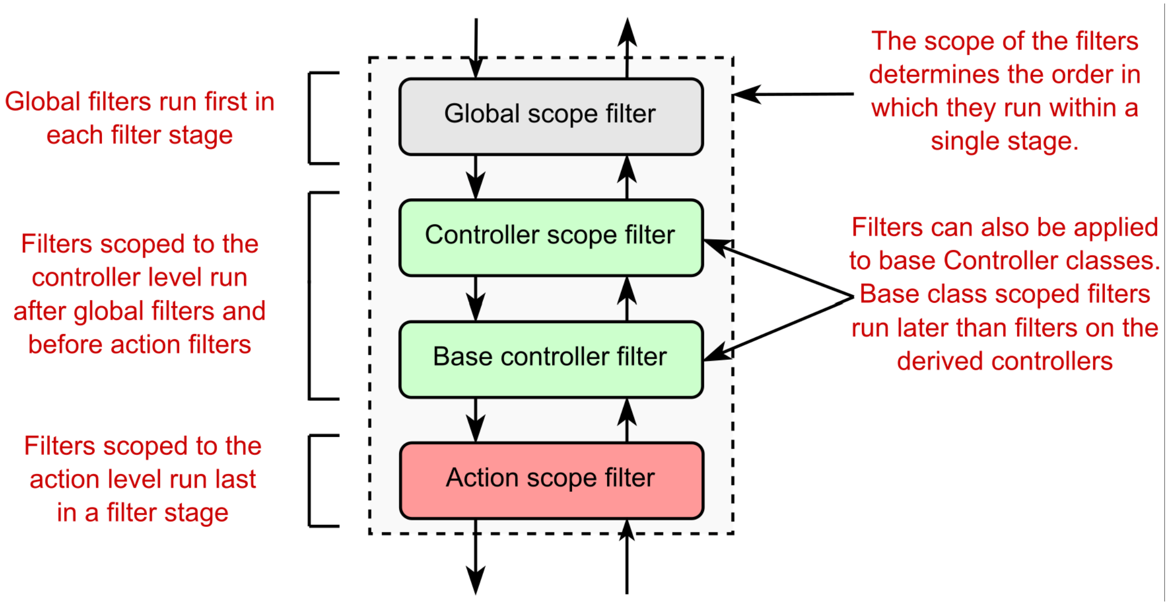

Global filters run first in each filter stage
Filters scoped to the controller level run after global filters and before action filters
Filters scoped to the action level run last in a filter stage
Global scope filter
Controller scope filter
Base controller filter
Action scope filter
The scope of the filters determines the order in which they run within a single stage.
Filters can also be applied to base Controller classes. Base class scoped filters run later than filters on the derived controllers

Figure 13.4 The default filter ordering within a given stage, based on the scope of the filters. For the `*Executing` method, globally scoped filters run first, followed by controller-scoped, and finally, action-scoped filters. For the `*Executed` method, the filters run in reverse order.

By default, filters execute from the broadest scope (global) to the narrowest (action) when running the `*Executing` method for each stage. The `*Executed` methods run in reverse order, from the narrowest scope (action) to the broadest (global).

The ordering for Razor Pages is somewhat simpler, given that you only have two scopes—global scope filters and Razor Page scope filters. For Razor Pages, global scope filters run the `*Executing` and `PageHandlerSelected` methods first, followed by the page scope filters. For the `*Executed` methods, the filters run in reverse order.

You'll sometimes find you need a bit more control over this order, especially if you have, for example, multiple action filters applied at the same scope. The filter pipeline caters to this requirement by way of the `IOrderedFilter` interface.

OVERRIDING THE DEFAULT ORDER OF FILTER EXECUTION WITH IORDEREDFILTER

Filters are great for extracting crosscutting concerns from your controller actions and Razor Page, but if you have multiple filters applied to an action, then you'll often need to control the precise order in which they execute.

Scope can get you some of the way, but for those other cases, you can implement `IOrderFilter`. This interface consists of a single property, `Order`:

```
public interface IOrderedFilter
{
    int Order { get; }
}
```

You can implement this property in your filters to set the order in which they execute. The filter pipeline orders the filters in a stage based on this value first, from lowest to highest, and uses the default scope order to handle ties, as shown in figure 13.5.

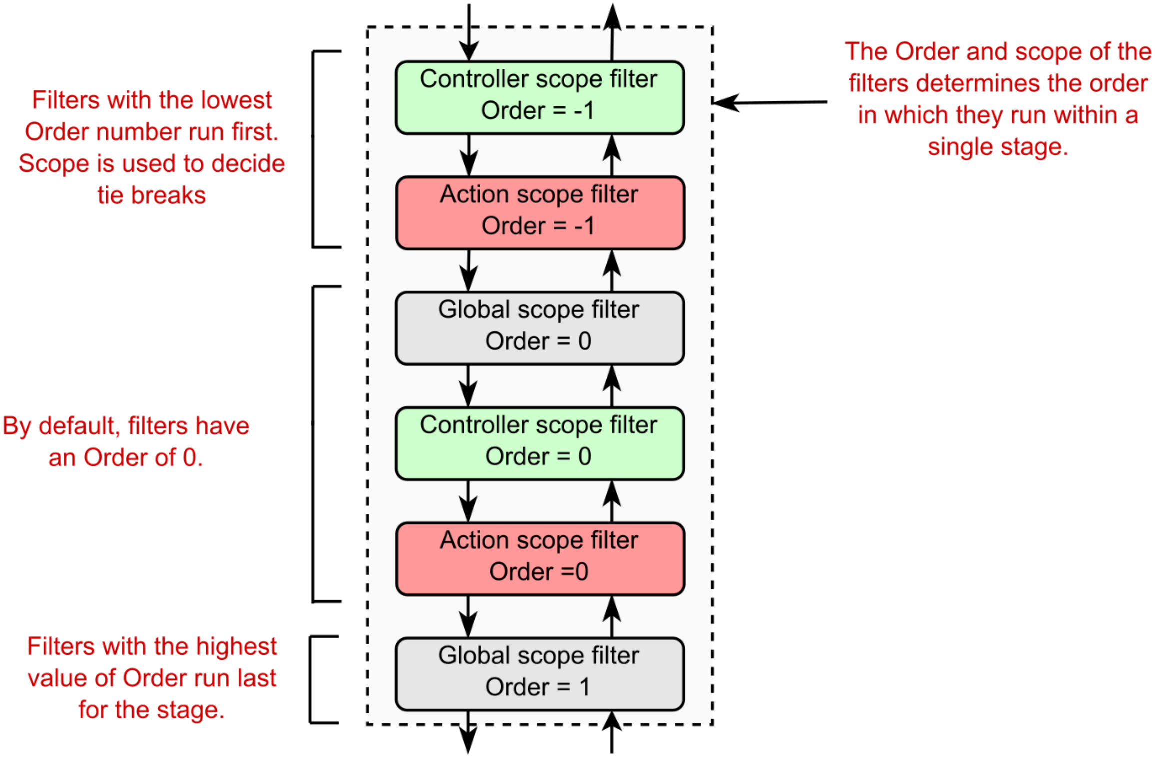

Filters with the lowest Order number run first. Scope is used to decide tie breaks
By default, filters have an Order of 0.
Filters with the highest value of Order run last for the stage.
Controller scope filter Order = -1
Action scope filter Order = -1
Global scope filter Order = 0
Controller scope filter Order = 0
Action scope filter Order =0
Global scope filter Order = 1
The Order and scope of the filters determines the order in which they run within a single stage.

Figure 13.5 Controlling the filter order for a stage using the `IOrderedFilter` interface. Filters are ordered by the `Order` property first, and then by scope.

The filters for `Order = -1` execute first, as they have the lowest `Order` value. The controller filter executes first because it has a broader scope than the action-scope filter. The filters with `Order=0` execute next, in the default scope order, as shown in figure 13.5. Finally, the filter with `Order=1` executes.

By default, if a filter doesn't implement `IOrderedFilter`, it's assumed to have `Order = 0`. All the filters that ship as part of ASP.NET Core have `Order = 0`, so you can implement your own filters relative to these.

This section has covered most of the technical details you need to use filters and create custom implementations for your own application. In the next section, you'll see some of the built-in filters provided by ASP.NET Core, as well as some practical examples of filters you might want to use in your own applications.

13.2 Creating custom filters for your application

ASP.NET Core includes a number of filters that you can use, but often, the most useful filters are the custom ones that are specific to your own apps. In this section, you'll work through each of the six types of filters. I'll explain in more detail what they're for and when you should use them. I'll point out examples of these filters that are part of ASP.NET Core itself and you'll see how to create custom filters for an example application.

To give you something realistic to work with, you'll start with a Web API controller for accessing the recipe application from chapter 12. This controller contains two actions: one for fetching a `RecipeDetailViewModel` and another for updating a `Recipe` with new values. This listing shows your starting point for this chapter, including both of the action methods.

Listing 13.8 Recipe Web API controller before refactoring to use filters

```
[Route("api/recipe")]
public class RecipeApiController : ControllerBase
{
    private const bool IsEnabled = true;           #A
    public RecipeService _service;
    public RecipeApiController(RecipeService service)
    {
        _service = service;
    }

    [HttpGet("{id}")]
    public IActionResult Get(int id)
    {
        if (!IsEnabled) { return BadRequest(); }   #B
        try
        {
            if (!_service.DoesRecipeExist(id))     #C
            {                                      #C
                return NotFound();                 #C
            }                                      #C
            var detail = _service.GetRecipeDetail(id);    #D
            Response.GetTypedHeaders().LastModified =     #E
                detail.LastModified;                      #E
            return Ok(detail);                            #F
        }
        catch (Exception ex)                 #G
        {                                    #G
            return GetErrorResponse(ex);     #G
        }                                    #G
    }

    [HttpPost("{id}")]
    public IActionResult Edit(
        int id, [FromBody] UpdateRecipeCommand command)
    {
        if (!IsEnabled) { return BadRequest(); }     #H
        try
        {
            if (!ModelState.IsValid)               #I
            {                                      #I
                return BadRequest(ModelState);     #I
            }                                      #I
            if (!_service.DoesRecipeExist(id))     #J
            {                                      #J
                return NotFound();                #J
            }                                     #J
            _service.UpdateRecipe(command);     #K
            return Ok();                        #K
        }
        catch (Exception ex)                 #L
        {                                    #L
            return GetErrorResponse(ex);     #L
        }                                    #L
    }

    private static IActionResult GetErrorResponse(Exception ex)
    {
        var error = new ProblemDetails
        {
            Title = "An error occured",
            Detail = context.Exception.Message,
            Status = 500,
            Type = "https://httpstatuses.com/500"
        };

        return new ObjectResult(error)
        {
            StatusCode = 500
        };
    }
}
```

#A This field would be passed in as configuration and is used to control access to actions.
#B If the API isn't enabled, block further execution.
#C If the requested Recipe doesn't exist, return a 404 response.
#D Fetch RecipeDetailViewModel.
#E Sets the Last-Modified response header to the value in the model
#F Returns the view model with a 200 response
#G If an exception occurs, catch it, and return the error in an expected format, as a 500 error.
#H If the API isn't enabled, block further execution.
#I Validates the binding model and returns a 400 response if there are errors
#J If the requested Recipe doesn't exist, return a 404 response.
#K Updates the Recipe from the command and returns a 200 response
#L If an exception occurs, catch it, and return the error in an expected format, as a 500 error.

These action methods currently have a *lot* of code to them, which hides the intent of each action. There's also quite a lot of duplication between the methods, such as checking that the `Recipe` entity exists, and formatting exceptions.

In this section, you’re going to refactor this controller to use filters for all the code in the methods that’s unrelated to the intent of each action. By the end of the chapter, you’ll have a much simpler controller that’s far easier to understand, as shown here.

Listing 13.9 Recipe Web API controller after refactoring to use filters

```
[Route("api/recipe")]
[ValidateModel, HandleException, FeatureEnabled(IsEnabled = true)]    #A
public class RecipeApiController : ControllerBase
{
    public RecipeService _service;
    public RecipeApiController(RecipeService service)
    {
        _service = service;
    }

    [HttpGet("{id}"), EnsureRecipeExists, AddLastModifedHeader]     #B
    public IActionResult Get(int id)
    {
        var detail = _service.GetRecipeDetail(id);      #C
        return Ok(detail);                              #C
    }

    [HttpPost("{id}"), EnsureRecipeExists]      #D
    public IActionResult Edit(
        int id, [FromBody] UpdateRecipeCommand command)
    {
        _service.UpdateRecipe(command);     #E
        return Ok();                        #E
    }
}
```

#A The filters encapsulate the majority of logic common to multiple action methods.
#B Placing filters at the action level limits them to a single action.
#C The intent of the action, return a Recipe view model, is much clearer.
#D Placing filters at the action level can be used to control the order in which they execute.
#E The intent of the action, update a Recipe, is much clearer.

I think you'll have to agree, the controller in listing 13.9 is much easier to read! In this section, you’ll refactor the controller bit by bit, removing crosscutting code to get to something more manageable. All the filters I’ll create in this section will use the sync filter interfaces—I’ll leave it as an exercise for the reader to create their async counterparts. You’ll start by looking at authorization filters and how they relate to security in ASP.NET Core.

13.2.1 Authorization filters: protecting your APIs

Authentication and *authorization* are related, fundamental concepts in security that we’ll be looking at in detail in chapters 14 and 15.

> **DEFINITION** *Authentication* is concerned with determining *who* made a request. *Authorization* is concerned with *what* a user is allowed to access.

Authorization filters run first in the MVC filter pipeline, before any other filters. They control access to the action method by immediately short-circuiting the pipeline when a request doesn't meet the necessary requirements.

ASP.NET Core has a built-in authorization framework that you should use when you need to protect your MVC application or your Web APIs. You can configure this framework with custom policies that let you finely control access to your actions.

> **TIP** It's possible to write your own authorization filters by implementing `IAuthorizationFilter` or `IAsyncAuthorizationFilter`, but I strongly advise against it. The ASP.NET Core authorization framework is highly configurable and should meet all your needs.

At the heart of the ASP.NET Core authorization framework is an Authorization filter, `AuthorizeFilter`, which you can add to the filter pipeline by decorating your actions or controllers with the `[Authorize]` attribute. In its simplest form, adding the `[Authorize]` attribute to an action, as in the following listing, means the request must be made by an authenticated user to be allowed to continue. If you're not logged in, it will short-circuit the pipeline, returning a `401 Unauthorized` response to the browser.

Listing 13.10 Adding `[Authorize]` to an action method

```
public class RecipeApiController : ControllerBase
{
    public IActionResult Get(int id)        #A
    {
        // method body
    }

    [Authorize]                             #B
    public IActionResult Edit(                             #C
        int id, [FromBody] UpdateRecipeCommand command)    #C
    {
        // method body
    }
}
```

#A The Get method has no [Authorize] attribute, so can be executed by anyone.
#B Adds the AuthorizeFilter to the filter pipeline using [Authorize]
#C The Edit method can only be executed if you're logged in.

As with all filters, you can apply the `[Authorize]` attribute at the controller level to protect all the actions on a controller, to a Razor Page to protect all the page handler methods in a page, or even globally, to protect every endpoint in your app.

> **NOTE** We'll explore authorization in detail in chapter 15, including how to add more detailed requirements, so that only specific sets of users can execute an action.

The next filters in the pipeline are resource filters. In the next section, you'll extract some of the common code from `RecipeApiController` and see how easy it is to create a short-circuiting filter.

13.2.2 Resource filters: short-circuiting your action methods

Resource filters are the first general-purpose filters in the MVC filter pipeline. In section 13.1.4, you saw minimal examples of both sync and async resource filters, which logged to the console. In your own apps, you can use resource filters for a wide range of purposes, thanks to the fact that they execute so early (and late) in the filter pipeline.

The ASP.NET Core framework includes a few different implementations of resource filters you can use in your apps, for example:

- `ConsumesAttribute`—Can be used to restrict the allowed formats an action method can accept. If your action is decorated with `[Consumes("application/json")]` but the client sends the request as XML, then the resource filter will short-circuit the pipeline and return a `415 Unsupported Media Type` response.
- `DisableFormValueModelBindingAttribute`—This filter prevents model binding from binding to form data in the request body. This can be useful if you know an action method will be handling large file uploads that you need to manage manually yourself. The resource filters run before model binding, so you can disable the model binding for a single action in this way.[59]

Resource filters are useful when you want to ensure the filter runs early in the pipeline, before model binding. They provide an early hook into the pipeline for your logic, so you can quickly short-circuit the request if you need to.

Look back at listing 13.8 and see if you can refactor any of the code into a Resource filter. One candidate line appears at the start of both the `Get` and `Edit` methods:

```
if (!IsEnabled) { return BadRequest(); }
```

This line of code is a *feature toggle* that you can use to disable the availability of the whole API, based on the `IsEnabled` field. In practice, you'd probably load the `IsEnabled` field from a database or configuration file so you could control the availability dynamically at runtime but, for this example, I'm using a hardcoded value.[60]

This piece of code is self-contained, crosscutting logic, which is somewhat tangential to the main intent of each action method—a perfect candidate for a filter. You want to execute the feature toggle early in the pipeline, before any other logic, so a resource filter makes sense.

[59] For details on handling file uploads, see https://docs.microsoft.com/aspnet/core/mvc/models/file-uploads.

[60] To read more about using feature toggles in your applications, see https://andrewlock.net/series/adding-feature-flags-to-an-asp-net-core-app/.

TIP Technically, you could also use an Authorization filter for this example, but I'm following my own advice of "Don't write your own Authorization filters!"

The next listing shows an implementation of `FeatureEnabledAttribute`, which extracts the logic from the action methods and moves it into the filter. I've also exposed the `IsEnabled` field as a property on the filter.

Listing 13.11 The `FeatureEnabledAttribute` resource filter

```
public class FeatureEnabledAttribute : Attribute, IResourceFilter
{
    public bool IsEnabled { get; set; }         #A
    public void OnResourceExecuting(            #B
        ResourceExecutingContext context)       #B
    {
        if (!IsEnabled)                                   #C
        {                                                 #C
            context.Result = new BadRequestResult();      #C
        }                                                 #C
    }
    public void OnResourceExecuted(              #D
        ResourceExecutedContext context) { }     #D
}
```

#A Defines whether the feature is enabled
#B Executes before model binding, early in the filter pipeline
#C If the feature isn't enabled, short-circuits the pipeline by setting the context.Result property
#D Must be implemented to satisfy IResourceFilter, but not needed in this case.

This simple resource filter demonstrates a number of important concepts, which are applicable to most filter types:

- The filter is an attribute as well as a filter. This lets you decorate your controller, action methods, and Razor Pages with it using `[FeatureEnabled(IsEnabled = true)]`.
- The filter interface consists of two methods—`*Executing` that runs before model binding and `*Executed` that runs after the result has been executed. You must implement both, even if you only need one for your use case.
- The filter execution methods provide a `context` object. This provides access to, among other things, the `HttpContext` for the request and metadata about the action method the middleware will execute.
- To short-circuit the pipeline, set the `context.Result` property to an `IActionResult` instance. The framework will execute this result to generate the response, bypassing any remaining filters in the pipeline, and skipping the action method (or page handler) entirely. In this example, if the feature isn't enabled, you bypass the pipeline by returning `BadRequestResult`, which will return a 400 error to the client.

By moving this logic into the resource filter, you can remove it from your action methods, and instead decorate the whole API controller with a simple attribute:

```
[Route("api/recipe"), FeatureEnabled(IsEnabled = true)]
public class RecipeApiController : ControllerBase
```

You've only extracted two lines of code from your action methods so far, but you're on the right track. In the next section, we'll move on to Action filters and extract two more filters from the action method code.

13.2.3 Action filters: customizing model binding and action results

Action filters run just after model binding, before the action method executes. Thanks to this positioning, action filters can access all the arguments that will be used to execute the action method, which makes them a powerful way of extracting common logic out of your actions.

On top of this, they also run just after the action method has executed and can completely change or replace the `IActionResult` returned by the action if you want. They can even handle exceptions thrown in the action.

> **REMINDER** Action filters don't execute for Razor Pages. Similarly, page filters don't execute for action methods.

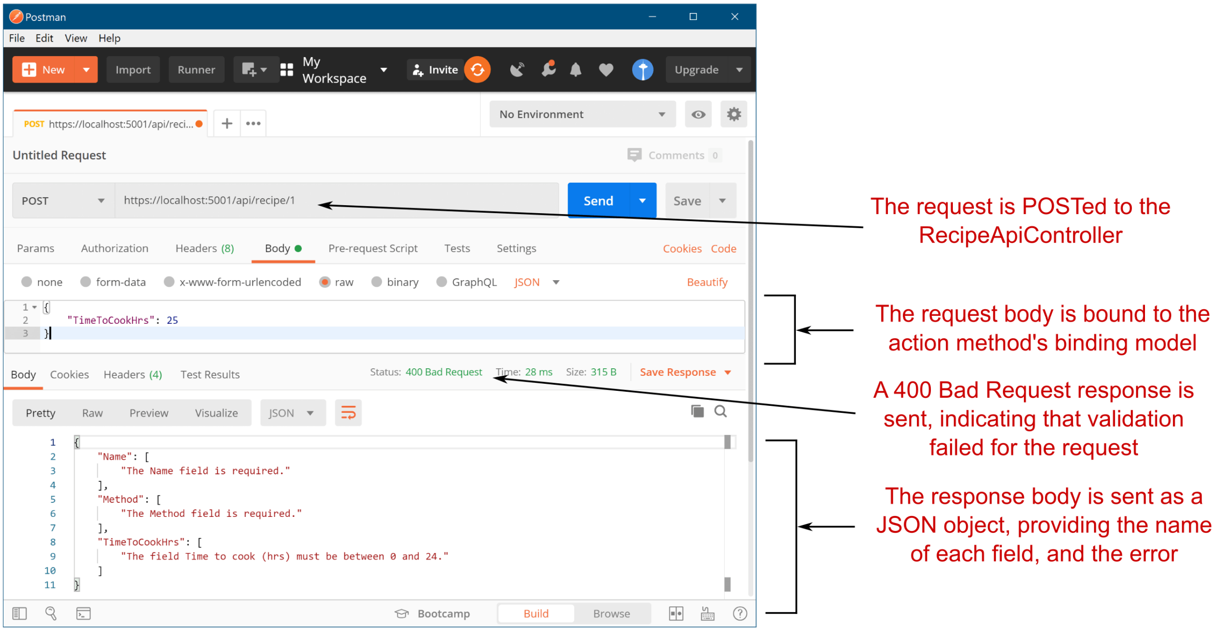
The ASP.NET Core framework includes several action filters out of the box. One of these commonly used filters is `ResponseCacheFilter`, which sets HTTP caching headers on your action-method responses.

> **TIP** Caching is a broad topic that aims to improve the performance of an application over the naive approach. But caching can also make debugging issues difficult and may even be undesirable in some situations. Consequently, I often apply `ResponseCacheFilter` to my action methods to set HTTP caching headers that *disable* caching! You can read about this and other approaches to caching in the docs at https://docs.microsoft.com/aspnet/core/performance/caching/response.

The real power of action filters comes when you build filters tailored to your own apps by extracting common code from your action methods. To demonstrate, I'm going to create two custom filters for `RecipeApiController`:

- `ValidateModelAttribute`—This will return `BadRequestResult` if the model state indicates that the binding model is invalid and will short-circuit the action execution. This attribute used to be a staple of my Web API applications, but the `[ApiController]` attribute now handles this (and more) for you. Nevertheless, I think it's useful to understand what's going on behind the scenes!
- `EnsureRecipeExistsAttribute`—This will use each action method's `id` argument to validate that the requested `Recipe` entity exists before the action method runs. If the `Recipe` doesn't exist, the filter will return `NotFoundResult` and will short-circuit the pipeline.

As you saw in chapter 6, the MVC framework automatically validates your binding models before executing your actions, but it's up to you to decide what to do about it. For Web API controllers, it’s common to return a 400 Bad Request response containing a list of the errors, as shown in figure 13.6.



POST https://localhost:5001/api/recipe/1

```
{
    "TimeToCookHrs": 25
}
```

Status: 400 Bad Request Time: 28 ms Size: 315 B

```
{
    "Name": [
        "The Name field is required."
    ],
    "Method": [
        "The Method field is required."
    ],
    "TimeToCookHrs": [
        "The field Time to cook (hrs) must be between 0 and 24."
    ]
}
```

The request is POSTed to the RecipeApiController

The request body is bound to the action method's binding model

A 400 Bad Request response is sent, indicating that validation failed for the request

The response body is sent as a JSON object, providing the name of each field, and the error

Figure 13.6 Posting data to a Web API using Postman. The data is bound to the action method’s binding model and validated. If validation fails, it’s common to return a 400 BadRequest response with a list of the validation errors.

You should ordinarily use the [ApiController] attribute on your Web API controllers, which gives you this behavior automatically. But if you can’t, or don’t want to, use that attribute, you can create a custom action filter instead. Listing 13.12 shows a basic implementation that is similar to the behavior you get with the [ApiController] attribute.

Listing 13.12 The action filter for validating `ModelState`

```
public class ValidateModelAttribute : ActionFilterAttribute     #A
{
    public override void OnActionExecuting(         #B
        ActionExecutingContext context)             #B
    {
        if (!context.ModelState.IsValid)            #C
        {
            context.Result =                                    #D
                new BadRequestObjectResult(context.ModelState);    #D
        }
    }
}
```

#A For convenience, you derive from the ActionFilterAttribute base class.
#B Overrides the Executing method to run the filter before the Action executes
#C Model binding and validation have already run at this point, so you can check the state.
#D If the model isn't valid, set the Result property; this short-circuits the action execution.

This attribute is self-explanatory and follows a similar pattern to the resource filter in section 13.2.2, but with a few interesting points:

- I have derived from the abstract `ActionFilterAttribute`. This class implements `IActionFilter` and `IResultFilter`, as well as their async counterparts, so you can override the methods you need as appropriate. This avoids needing to add an unused `OnActionExecuted()` method, but using the base class is entirely optional and a matter of preference.
- Action filters run after model binding has taken place, so `context.ModelState` contains the validation errors if validation failed.
- Setting the `Result` property on `context` short-circuits the pipeline. But, due to the position of the action filter stage, only the action method execution and later action filters are bypassed; all the other stages of the pipeline run as though the action had executed as normal.

If you apply this action filter to your `RecipeApiController`, you can remove

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

from the start of both the action methods, as it will run automatically in the filter pipeline. You'll use a similar approach to remove the duplicate code checking whether the `id` provided as an argument to the action methods corresponds to an existing Recipe entity.

This listing shows the `EnsureRecipeExistsAttribute` action filter. This uses an instance of `RecipeService` to check whether the `Recipe` exists and returns a 404 Not Found if it doesn't.

Listing 13.13 An action filter to check whether a Recipe exists

```
public class EnsureRecipeExistsAtribute : ActionFilterAttribute
{
    public override void OnActionExecuting(
        ActionExecutingContext context)
    {
        var service = (RecipeService) context.HttpContext        #A
            .RequestServices.GetService(typeof(RecipeService));  #A
        var recipeId = (int) context.ActionArguments["id"];      #B
        if (!service.DoesRecipeExist(recipeId))                  #C
        {
            context.Result = new NotFoundResult();               #D
        }
    }
}
```

#A Fetches an instance of RecipeService from the DI container
#B Retrieves the id parameter that will be passed to action method when it executes
#C Checks whether a Recipe entity with the given RecipeId exists
#D If it doesn't exist, returns a 404 Not Found result and short-circuits the pipeline.

As before, you've derived from `ActionFilterAttribute` for simplicity and overridden the `OnActionExecuting` method. The main functionality of the filter relies on the `DoesRecipeExist()` method of `RecipeService`, so the first step is to obtain an instance of `RecipeService`. The `context` parameter provides access to the `HttpContext` for the request, which in turn lets you access the DI container and use `RequestServices.GetService()` to return an instance of `RecipeService`.

> **WARNING** This technique for obtaining dependencies is known as *service location* and is generally considered an antipattern.[61] In section 13.4, I'll show a better way to use the DI container to inject dependencies into your filters.

As well as `RecipeService`, the other piece of information you need is the `id` argument of the `Get` and `Edit` action methods. In action filters, model binding has already occurred, so the arguments that the framework will use to execute the action method are already known and are exposed on `context.ActionArguments`.

The action arguments are exposed as `Dictionary<string, object>`, so you can obtain the `id` parameter using the `"id" string` key. Remember to cast the object to the correct type.

> **TIP** Whenever I see magic strings like this, I always like to try to replace them by using the `nameof` operator. Unfortunately, `nameof` won't work for method arguments like this, so be careful when refactoring your code. I suggest explicitly applying the action filter to the action method (instead of globally, or to a controller) to remind you about that implicit coupling.

With `RecipeService` and `id` in place, it's a case of checking whether the identifier corresponds to an existing `Recipe` entity and, if not, setting `context.Result` to `NotFoundResult`. This short-circuits the pipeline and bypasses the action method altogether.

> **NOTE** Remember, you can have multiple action filters running in a single stage. Short-circuiting the pipeline by setting `context.Result` will prevent later filters in the stage from running, as well as bypassing the action method execution.

[61] For a detailed discussion on DI patterns and antipatterns, see *Dependency Injection Principles, Practices, and Patterns* by Steven van Deursen and Mark Seemann (Manning, 2019) https://livebook.manning.com/book/dependency-injection-principles-practices-patterns/chapter-5.

Before we move on, it’s worth mentioning a special case for action filters. The `ControllerBase` base class implements `IActionFilter` and `IAsyncActionFilter` itself. If you find yourself creating an action filter for a single controller and you want to apply it to every action in that controller, then you can override the appropriate methods on your controller.

Listing 13.14 Overriding action filter methods directly on `ControllerBase`

```
public class HomeController : ControllerBase       #A
{
    public override void OnActionExecuting(      #B
        ActionExecutingContext context)          #B
    { }                                          #B
    public override void OnActionExecuted(       #C
        ActionExecutedContext context)           #C
    { }                                          #C
}
```

#A Derives from the ControllerBase class
#B Runs before any other action filters for every action in the controller
#C Runs after all other action filters for every action in the controller

If you override these methods on your controller, they’ll run in the action filter stage of the filter pipeline for every action on the controller. The `OnActionExecuting` `ControllerBase` method runs before any other action filters, regardless of ordering or scope, and the `OnActionExecuted` method runs after all other action filters.

> **TIP** **The controller implementation can be useful in some cases, but you can’t control the ordering related to other filters. Personally, I generally prefer to break logic into explicit, declarative filter attributes but, as always, the choice is yours.**

With the resource and action filters complete, your controller is looking much tidier, but there’s one aspect in particular that would be nice to remove: the exception handling. In the next section, we’ll look at how to create a custom exception filter for your controller, and why you might want to do this instead of using exception-handling middleware.

13.2.4 Exception filters: custom exception handling for your action methods

In chapter 3, I went into some depth about types of error-handling middleware you can add to your apps. These let you catch exceptions thrown from any later middleware and handle them appropriately. If you’re using exception-handling middleware, you may be wondering why we need exception filters at all!

The answer to this is pretty much the same as I outlined in section 13.1.3: filters are great for crosscutting concerns, when you need behavior that’s either specific to MVC or should only apply to certain routes.

Both of these can apply in exception handling. Exception filters are part of the MVC framework, so they have access to the context in which the error occurred, such as the action or Razor Page that was executing. This can be useful for logging additional details when errors occur, such as the action parameters that caused the error.

> **WARNING** **If you use exception filters to record action method arguments, make sure you're not storing sensitive data, such as passwords or credit card details, in your logs.**

You can also use exception filters to handle errors from different routes in different ways. Imagine you have both Razor Pages and Web API controllers in your app, as we do in the recipe app. What happens when an exception is thrown by a Razor Page?

As you saw in chapter 3, the exception travels back up the middleware pipeline, and is caught by exception-handler middleware. The exception-handler middleware will re-execute the pipeline and generate an HTML error page.

That's great for your Razor Pages, but what about exceptions in your Web API controllers? If your API throws an exception, and consequently returns HTML generated by the exception-handler middleware, that's going to break a client who has called the API expecting a JSON response!

Instead, exception filters let you handle the exception in the filter pipeline and generate an appropriate response body. The exception handler middleware only intercepts errors without a body, so it will let the modified Web API response pass untouched.

> **NOTE** **The `[ApiController]` attribute converts error `StatusCodeResult`s to a `ProblemDetails` object, but it *doesn't* catch exceptions.**

Exception filters can catch exceptions from more than just your action methods and page handlers. They'll run if an exception occurs:

- During model binding or validation
- When the action method or page handler is executing
- When an action filter or page filter is executing

You should note that exception filters won't catch exceptions thrown in any filters other than action and page filters, so it's important your resource and result filters don't throw exceptions. Similarly, they won't catch exceptions thrown when executing an `IActionResult`, for example when rendering a Razor view to HTML.

Now that you know why you might want an exception filter, go ahead and implement one for `RecipeApiController`, as shown next. This lets you safely remove the `try-catch` block from your action methods, knowing that your filter will catch any errors.

Listing 13.15 The `HandleExceptionAttribute` exception filter

```
public class HandleExceptionAttribute : ExceptionFilterAttribute      #A
{
    public override void OnException(ExceptionContext context)        #B
    {
        var error = new ProblemDetails                   #C
```

©Manning Publications Co. To comment go to liveBook

```
        {                                           #C
            Title = "An error occured",             #C
            Detail = context.Exception.Message,     #C
            Status = 500,                           #C
            Type = "https://httpstatuses.com/500"     #C
        };                                          #C

        context.Result = new ObjectResult(error)   #D
        {                                          #D
            StatusCode = 500                       #D
        };                                         #D
        context.ExceptionHandled = true;           #E
    }
}
```

#A ExceptionFilterAttibute is an abstract base class that implements IExceptionFilter
#B There's only a single method to override for IExceptionFilter
#C Building a problem details object to return in the response
#D Creates an ObjectResult to serialize the ProblemDetails and to set the response status code
#E Marks the exception as handled to prevent it propagating into the middleware pipeline

It's quite common to have an exception filter in your application, especially if you are mixing API controllers and Razor Pages in your application, but they're not always necessary. If you can handle all the exceptions in your application with a single piece of middleware, then ditch the exception filters and go with that instead.

You're almost done refactoring your `RecipeApiController`, you just have one more filter type to add: result filters. Custom result filters tend to be relatively rare in the apps I've written, but they have their uses, as you'll see.

13.2.5 Result filters: customizing action results before they execute

If everything runs successfully in the pipeline, and there's no short-circuiting, then the next stage of the pipeline after action filters are result filters. These run just before and after the `IActionResult` returned by the Action method (or action filters) is executed.

> **WARNING** If the pipeline is short-circuited by setting `context.Result`, the result filter stage won't be run, but `IActionResult` will still be executed to generate the response. The exceptions to this rule are action and page filters—these only short-circuit the action execution, as you saw in figures 13.2 and 13.3, and so result filters run as normal, as though the action or page handler itself generated the response.

Result filters run immediately after action filters, so many of their use-cases are similar, but you typically use result filters to customize the way the `IActionResult` executes. For example, ASP.NET Core has several result filters built into its framework:

- `ProducesAttribute`—This forces a Web API result to be serialized to a specific output format. For example, decorating your action method with `[Produces("application/xml")]` forces the formatters to try to format the response as XML, even if the client doesn't list XML in its `Accept` header.

- FormatFilterAttribute—Decorating an action method with this filter tells the formatter to look for a route value or query string parameter called format, and to use that to determine the output format. For example, you could call /api/recipe/11?format=json and FormatFilter will format the response as JSON, or call api/recipe/11?format=xml and get the response as XML.[62]

As well as controlling the output formatters, you can use result filters to make any last-minute adjustments before IActionResult is executed and the response is generated.

As an example of the kind of flexibility available, in the following listing I demonstrate setting the LastModified header, based on the object returned from the action. This is a somewhat contrived example—it's specific enough to a single action that it doesn't warrant being moved to a result filter—but, hopefully, you get the idea!

Listing 13.16 Setting a response header in a result filter

```
public class AddLastModifedHeaderAttribute : ResultFilterAttribute    #A
{
    public override void OnResultExecuting(         #B
        ResultExecutingContext context)             #B
    {
        if (context.Result is OkObjectResult result            #C
            && result.Value is RecipeDetailViewModel detail)   #D
        {
            var viewModelDate = detail.LastModified;             #E
            context.HttpContext.Response                         #E
              .GetTypedHeaders().LastModified = viewModelDate;   #E
        }
    }
}
```

#A ResultFilterAttribute provides a useful base class you can override
#B You could also override the Executed method, but the response would already be sent by then
#C Checks whether the action result returned a 200 Ok result with a view model
#D Checks whether the view model type is RecipeDetailViewModel . . .
#E . . . if it is, fetches the LastModified property and sets the Last-Modified header in the response

I've used another helper base class here, ResultFilterAttribute, so you only need to override a single method to implement the filter. Fetch the current IActionResult, exposed on context.Result, and check that it's an OkObjectResult instance with a RecipeDetailViewModel value. If it is, then fetch the LastModified field from the view model and add a Last-Modified header to the response.

[62] Remember, you need to explicitly configure the XML formatters if you want to serialize to XML. For details on formatting results based on the URL, see https://andrewlock.net/formatting-response-data-as-xml-or-json-based-on-the-url-in-asp-net-core/.

> **TIP** `GetTypedHeaders()` is an extension method that provides strongly typed access to request and response headers. It takes care of parsing and formatting the values for you. You can find it in the `Microsoft.AspNetCore.Http` namespace.

As with resource and action filters, result filters can implement a method that runs *after* the result has been executed, `OnResultExecuted`. You can use this method, for example, to inspect exceptions that happened during the execution of `IActionResult`.

> **WARNING** Generally, you can't modify the response in the `OnResultExecuted` method, as you may have already started streaming the response to the client.

We've finished simplifying the `RecipeApiController` now. By extracting various pieces of functionality to filters, the original controller in listing 13.8 can be simplified to the version in 13.9. This is obviously a somewhat extreme and contrived demonstration, and I'm not advocating that filters should always be your go-to option.

> **TIP** Filters should be a last resort in most cases. Where possible, it is often preferable to use a simple private method in a controller, or push functionality into the domain instead of using filters. Filters should generally be used to extract repetitive, HTTP-related, or common cross-cutting code from your controllers.

There's still one more filter we haven't looked at yet, because it only applies to Razor Pages: page filters.

13.2.6 Page filters: customizing model binding for Razor Pages

As already discussed, action filters only apply to controllers and actions; they have no effect on Razor Pages. Similarly, page filters have no effect on controllers and actions. Nevertheless, page filters and action filters fulfil similar roles.

Just like action filters, The ASP.NET Core framework includes several page filters out of the box. One of these is the Razor Page equivalent of the caching action filter, `ResponseCacheFilter`, called `PageResponseCacheFilter`. This works identically to the action-filter equivalent I described in section 13.2.3, setting HTTP caching headers on your Razor Page responses.

Page filters are somewhat unusual, as they implement three methods, as discussed in section 13.1.2. In practice, I've rarely seen a page filter that implements all three. It's unusual to need to run code immediately after page handler selection and before model validation. It's far more common to perform a role directly analogous to action filters.

For example, the following listing shows a page filter equivalent to the `EnsureRecipeExistsAttribute` action filter.

Listing 13.17 A page filter to check whether a `Recipe` exists

```
public class PageEnsureRecipeExistsAtribute : Attribute, IPageFilter    #A
{
    public void OnPageHandlerSelected(            #B
        PageHandlerSelectedContext context)       #B
    {}                                            #B

    public void OnPageHandlerExecuting(            #C
        PageHandlerExecutingContext context)       #C
    {
        var service = (RecipeService) context.HttpContext            #D
            .RequestServices.GetService(typeof(RecipeService));      #D
        var recipeId = (int) context.HandlerArguments["id"];         #E
        if (!service.DoesRecipeExist(recipeId))                      #F
        {
            context.Result = new NotFoundResult();                   #G
        }
    }

    public void OnPageHandlerExecuted(              #H
        PageHandlerExecutedContext context)         #H
    { }                                             #H
}
```

#A Implement IPageFilter and as an attribute so you can decorate the Razor Page PageModel
#B Executed after handler selection, before model binding. Not used in this example
#C Executed after model binding and validation, before page handler execution
#D Fetches an instance of RecipeService from the DI container
#E Retrieves the id parameter that will be passed to page handler method when it executes
#F Checks whether a Recipe entity with the given RecipeId exists
#G If it doesn't exist, returns a 404 Not Found result and short-circuits the pipeline
#H Executed after page handler execution (or short-circuiting). Not used in this example

The page filter is very similar to the action filter equivalent. The most obvious difference is the need to implement three methods to satisfy the `IPageFilter` interface. You'll commonly want to implement the `OnPageHandlerExecuting` method, which runs just after model binding and validation, and before the page handler executes.

A subtle difference between the action filter code and the page filter code is that the action filter accesses the model-bound action arguments using `context.ActionArguments`. The page filter uses `context.HandlerArguments` in the example, but there's also another option.

Remember from chapter 6 that Razor Pages often bind to public properties on the `PageModel` using the `[BindProperty]` attribute. You can access those properties directly, instead of having to use magic strings, by casting a `HandlerInstance` property to the correct `PageModel` type, and accessing the property directly. For example,

```
var recipeId = ((ViewRecipePageModel)context.HandlerInstance).Id
```

Just as the `ControllerBase` class implements `IActionFilter`, so `PageModel` implements `IPageFilter` and `IAsyncPageFilter`. If you want to create an action filter for a single Razor Page, then you could save yourself the trouble of creating a separate page filter, and override these methods directly in your Razor Page.

> **TIP** **I generally find it's not worth the hassle of using page filters unless you have a *very* common requirement. The extra level of indirection page filters add, coupled with the typically "bespoke" nature of individual Razor**

Pages, means I normally find they aren't worth using. Your mileage may vary of course, but don't jump to them as a first option!

That brings us to the end of this detailed look at each of the filters in the MVC pipeline. Looking back and comparing listings 13.8 and 13.9, you can see filters allowed us to refactor the controllers and make the intent of each action method much clearer. Writing your code in this way makes it easier to reason about, as each filter and action has a single responsibility.

In the next section, I'll take a slight detour into exactly what happens when you short-circuit a filter. I've described *how* to do this, by setting the `context.Result` property on a filter, but I haven't yet described exactly what happens. For example, what if there are multiple filters in the stage when it's short-circuited? Do those still run?

13.3 Understanding pipeline short-circuiting

In this short section you'll learn about the details of filter-pipeline short-circuiting. You'll see what happens to the other filters in a stage when the pipeline is short-circuited, and how to short-circuit each type of filter.

A brief warning: the topic of filter short-circuiting can be a little confusing. Unlike middleware short-circuiting, which is cut-and-dried, the filter pipeline is a bit more nuanced. Luckily, you won't often find you need to dig into it, but when you do, you'll be glad of the detail.

You short-circuit the authorization, resource, action, page, and result filters by setting `context.Result` to `IActionResult`. Setting an action result in this way causes some, or all, of the remaining pipeline to by bypassed. But the filter pipeline isn't entirely linear, as you saw in figures 13.2 and 13.3, so short-circuiting doesn't always do an about-face back down the pipeline. For example, short-circuited action filters only bypass action method execution—the result filters and result execution stages still run.

The other difficultly is what happens if you have more than one type of filter. Let's say you have three resource filters executing in a pipeline. What happens if the second filter causes a short circuit? Any remaining filters are bypassed, but the first resource filter has already run its `*Executing` command, as shown in figure 13.7. This earlier filter gets to run its `*Executed` command too, with `context.Cancelled = true`, indicating that a filter in that stage (the resource filter stage) short-circuited the pipeline.

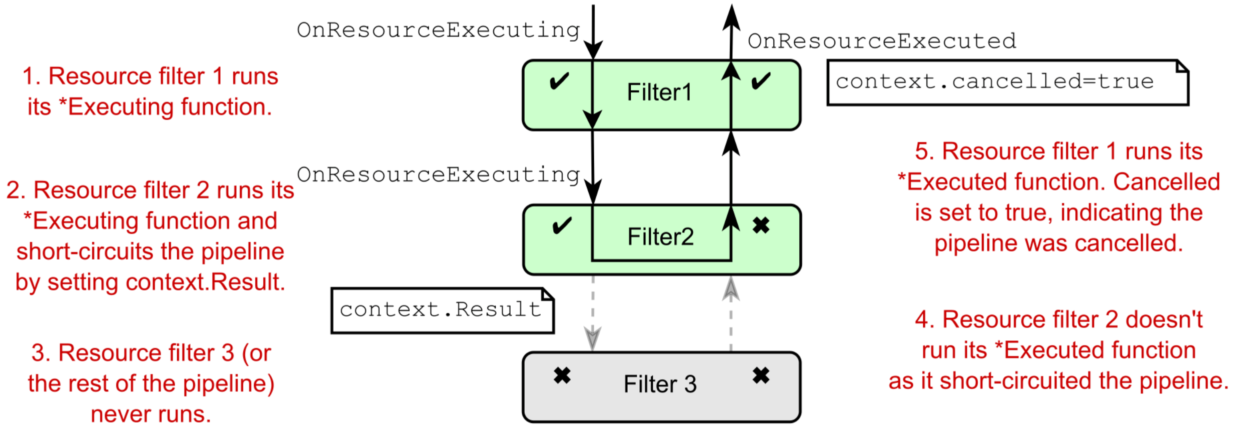

OnResourceExecuting
OnResourceExecuted
Filter1
✔
✔
context.cancelled=true
1. Resource filter 1 runs its *Executing function.
OnResourceExecuting
2. Resource filter 2 runs its *Executing function and short-circuits the pipeline by setting context.Result.
Filter2
✔
✖
5. Resource filter 1 runs its *Executed function. Cancelled is set to true, indicating the pipeline was cancelled.
context.Result
4. Resource filter 2 doesn't run its *Executed function as it short-circuited the pipeline.
3. Resource filter 3 (or the rest of the pipeline) never runs.
Filter 3
✖
✖

Figure 13.7 The effect of short-circuiting a resource filter on other resource filters in that stage. Later filters in the stage won't run at all, but earlier filters run their `OnResourceExecuted` function.

Understanding which other filters will run when you short-circuit a filter can be somewhat of a chore, but I've summarized each filter in table 13.1. You'll also find it useful to refer to figures 13.2 and 13.3 to visualize the shape of the pipeline when thinking about short-circuits.

Table 13.1 The effect of short-circuiting filters on filter-pipeline execution

Filter type	How to short-circuit?	What else runs?
Authorization filters	Set `context.Result`	Nothing, the pipeline is immediately halted.
Resource filters	Set `context.Result`	Resource-filter `*Executed` functions from earlier filters run with `context.Cancelled = true`.
Action filters	Set `context.Result`	Only bypasses action method execution. Action filters earlier in the pipeline run their `*Executed` methods with `context.Cancelled = true`, then result filters, result execution, and resource filters' `*Executed` methods all run as normal.
Page filters	Set `context.Result` in `OnPageHandlerSelected`	Only bypasses page handler execution. Page filters earlier in the pipeline run their `*Executed` methods with `context.Cancelled = true`, then result filters, result execution, and resource filters' `*Executed` methods all

		run as normal.
Exception filters	Set `context.Result` and `Exception.Handled = true`	All resource-filter *`Executed` functions run.
Result filters	Set `context.Cancelled = true`	Result filters earlier in the pipeline run their *`Executed` functions with `context.Cancelled = true`. All resource–filter *`Executed` functions run as normal.

The most interesting point here is that short-circuiting an action filter (or a page filter) doesn't short-circuit much of the pipeline at all. In fact, it only bypasses later action filters and the action method execution itself. By primarily building action filters, you can ensure that other filters, such as result filters that define the output format, run as usual, even when your action filters short-circuit.

The last thing I'd like to talk about in this chapter is how to use DI with your filters. You saw in chapter 10 that DI is integral to ASP.NET Core, and in the next section, you'll see how to design your filters so that the framework can inject service dependencies into them for you.

13.4 Using dependency injection with filter attributes

In this section you'll learn how to inject services into your filters, so you can take advantage of the simplicity of DI in your filters. You'll learn to use two helper filters to achieve this `TypeFilterAttribute` and `ServiceFilterAttribute`, and you'll see how they can be used to simplify the action filter you defined in section 13.2.3.

The previous version of ASP.NET used filters, but they suffered from one problem in particular: it was hard to use services from them. This was a fundamental issue with implementing them as attributes that you decorate your actions with. C# attributes don't let you pass dependencies into their constructors (other than constant values), and they're created as singletons, so there's only a single instance for the lifetime of your app.

In ASP.NET Core, this limitation is still there in general, in that filters are typically created as attributes that you add to your controller classes, action methods, and Razor Pages. What happens if you need to access a transient or scoped service from inside the singleton attribute?

Listing 13.13 showed one way of doing this, using a pseudo service locator pattern to reach into the DI container and pluck out `RecipeService` at runtime. This works but is generally frowned upon as a pattern, in favor of proper DI. How can you add DI to your filters?

The key is to split the filter into two. Instead of creating a class that's both an attribute and a filter, create a filter class that contains the functionality and an attribute that tells the framework when and where to use the filter.

Let's apply this to the action filter from listing 13.13. Previously, I derived from ActionFilterAttribute and obtained an instance of RecipeService from the context passed to the method. In the following listing, I show two classes, EnsureRecipeExistsFilter and EnsureRecipeExistsAttribute. The filter class is responsible for the functionality and takes in RecipeService as a constructor dependency.

Listing 13.18 Using DI in a filter by not deriving from Attribute

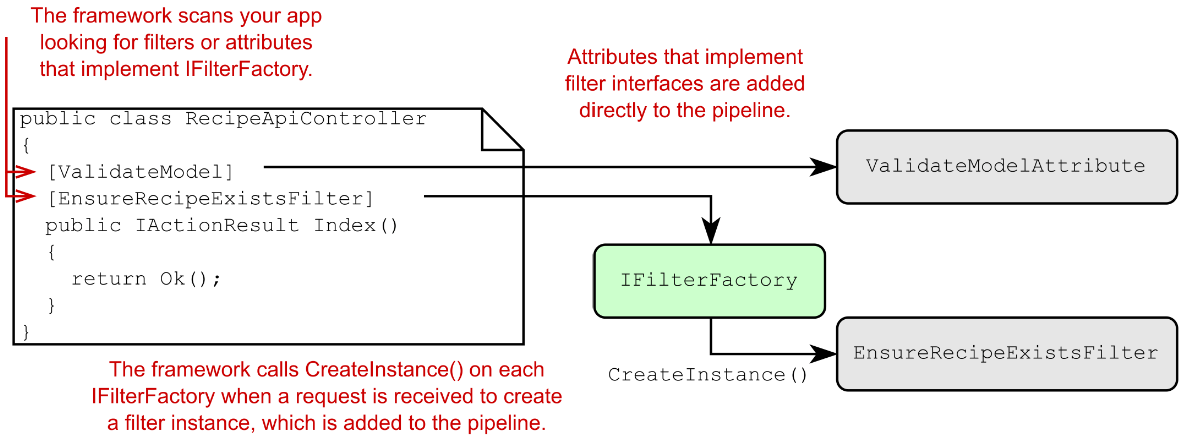
```
public class EnsureRecipeExistsFilter : IActionFilter            #A
{
    private readonly RecipeService _service;                    #B
    public EnsureRecipeExistsFilter(RecipeService service)      #B
    {                                                           #B
        _service = service;                                     #B
    }                                                           #B

    public void OnActionExecuting(ActionExecutingContext context)      #C
    {                                                                  #C
        var recipeId = (int) context.ActionArguments["id"];            #C
        if (!_service.DoesRecipeExist(recipeId))                       #C
        {                                                              #C
            context.Result = new NotFoundResult();                     #C
        }                                                              #C
    }                                                                  #C

    public void OnActionExecuted(ActionExecutedContext context) { }    #D
}

public class EnsureRecipeExistsAttribute : TypeFilterAttribute       #E
{
    public EnsureRecipeExistsAttribute()                             #F
        : base(typeof(EnsureRecipeExistsFilter)) {}                  #F
}
```

#A Doesn't derive from an Attribute class
#B RecipeService is injected into the constructor
#C The rest of the method remains the same
#D You must implement the Executed action to satisfy the interface
#E Derives from TypeFilter, which is used to fill dependencies using the DI container
#F Passes the type EnsureRecipeExistsFilter as an argument to the base TypeFilter constructor

EnsureRecipeExistsFilter is a valid filter; you could use it on its own by adding it as a global filter (as global filters don't need to be attributes). But you can't use it directly by decorating controller classes and action methods, as it's not an attribute. That's where EnsureRecipeExistsAttribute comes in.

You can decorate your methods with EnsureRecipeExistsAttribute instead. This attribute inherits from TypeFilterAttribute and passes the Type of filter to create as an argument to the base constructor. This attribute acts as a *factory* for EnsureRecipeExistsFilter by implementing IFilterFactory.

When ASP.NET Core initially loads your app, it scans your actions and controllers, looking for filters and filter factories. It uses these to form a filter pipeline for every action in your app, as shown in figure 13.8.



The framework scans your app looking for filters or attributes that implement IFilterFactory.

Attributes that implement filter interfaces are added directly to the pipeline.

```
public class RecipeApiController
{
    [ValidateModel]
    [EnsureRecipeExistsFilter]
    public IActionResult Index()
    {
        return Ok();
    }
}
```

The framework calls CreateInstance() on each IFilterFactory when a request is received to create a filter instance, which is added to the pipeline.

Figure 13.8 The framework scans your app on startup to find both filters and attributes that implement IFilterFactory. At runtime, the framework calls CreateInstance() to get an instance of the filter.

When an action decorated with EnsureRecipeExistsAttribute is called, the framework calls CreateInstance() on the attribute. This creates a new instance of EnsureRecipeExistsFilter and uses the DI container to populate its dependencies (RecipeService).

By using this IFilterFactory approach, you get the best of both worlds; you can decorate your controllers and actions with attributes, and you can use DI in your filters. Out of the box, two similar classes provide this functionality, which have slightly different behaviors:

- TypeFilterAttribute—Loads all of the filter's dependencies from the DI container and uses them to create a new instance of the filter.
- ServiceFilterAttribute—Loads the filter *itself* from the DI container. The DI container takes care of the service lifetime and building the dependency graph. Unfortunately, you also have to explicitly register your filter with the DI container in ConfigureServices in Startup:

```
services.AddTransient<EnsureRecipeExistsFilter>();
```

Whether you choose to use TypeFilterAttribute or ServiceFilterAttribute is somewhat a matter of preference, and you can always implement a custom IFilterFactory if you need to. The key takeaway is that you can now use DI in your filters. If you don't need to use DI for a filter, then implement it as an attribute directly for simplicity.

> **TIP** **I like to create my filters as a nested class of the attribute class when using this pattern. This keeps all the code nicely contained in a single file and indicates the relationship between the classes.**

That brings us to the end of this chapter on the filter pipeline. Filters are a somewhat advanced topic, in that they aren't strictly necessary for building basic apps, but I find them extremely useful for ensuring my controller and action methods are simple and easy to understand.

In the next chapter, we'll take our first look at securing your app. We'll discuss the difference between authentication and authorization, the concept of identity in ASP.NET Core, and how you can use the ASP.NET Core Identity system to let users register and log in to your app.

13.5 Summary

- The filter pipeline executes as part of the MVC or Razor Pages execution. It consists of authorization filters, resource filters, action filters, page filters, exception filters, and Result filters. Each filter type is grouped into a *stage* and can be used to achieve effects specific to that stage.
- Resource, action, and result filters run twice in the pipeline: an `*Executing` method on the way in and an `*Executed` method on the way out. Page filters run three times: after page handler selection, and before and after page handler execution.
- Authorization and exception filters only run once as part of the pipeline; they don't run after a response has been generated.
- Each type of filter has both a sync and an async version. For example, resource filters can implement either the `IResourceFilter` interface or the `IAsyncResourceFilter` interface. You should use the synchronous interface unless your filter needs to use asynchronous method calls.
- You can add filters globally, at the controller level, at the Razor Page level, or at the action level. This is called the *scope* of the filter. Which scope you should choose depends on how broadly you want to apply the filter.
- Within a given stage, global-scoped filters run first, then controller-scoped, and finally, action-scoped. You can also override the default order by implementing the `IOrderedFilter` interface. Filters will run from lowest to highest `Order` and use scope to break ties.
- Authorization filters run first in the pipeline and control access to APIs. ASP.NET Core includes an `[Authorization]` attribute that you can apply to action methods so that only logged-in users can execute the action.
- Resource filters run after authorization filters, and again after a result has been executed. They can be used to short-circuit the pipeline, so that an action method is

never executed. They can also be used to customize the model binding process for an action method.

- Action filters run after model binding has occurred, just before an action method executes. They also run after the action method has executed. They can be used to extract common code out of an action method to prevent duplication. They don't execute for Razor Pages, only for MVC controllers.
- The `ControllerBase` base class also implements `IActionFilter` and `IAsyncActionFilter`. They run at the start and end of the action filter pipeline, regardless of the ordering or scope of other action filters. They can be used to create action filters that are specific to one controller.
- Page filters run three times: after page handler selection, after model binding, and after the page handler method executes. You can use page filters for similar purposes as action filters. Page filters only execute for Razor Pages, they don't run for MVC controllers.
- Razor Page `PageModels` implement `IPageFilter` and `IAsyncPageFilter`, so can be used to implement page-specific page filters. These are rarely used, as you can typically achieve similar results with simple private methods.
- Exception filters execute after action and page filters, when an action method or page handler has thrown an exception. They can be used to provide custom error handling specific to the action executed.
- Generally, you should handle exceptions at the middleware level, but you can use exception filters to customize how you handle exceptions for specific actions, controllers, or Razor Pages.
- Result filters run just before and after an `IActionResult` is executed. You can use them to control how the action result is executed, or to completely change the action result that will be executed.
- You can use `ServiceFilterAttribute` and `TypeFilterAttribute` to allow dependency injection in your custom filters. `ServiceFilterAttribute` requires that you register your filter and all its dependencies with the DI container, whereas `TypeFilterAttribute` only requires that the filter's dependencies have been registered.

14

Authentication: adding users to your application with Identity

This chapter covers

- **How authentication works in web apps in ASP.NET Core**
- **Creating a project using the ASP.NET Core Identity system**
- **Adding user functionality to an existing web app**
- **Customizing the default ASP.NET Core Identity UI**

One of the selling points of a web framework like ASP.NET Core is the ability to provide a dynamic app, customized to individual users. Many apps have the concept of an "account" with the service, which you can "sign in" to and get a different experience.

Depending on the service, an account gives you varying things: on some apps you may have to sign in to get access to additional features, on others you might see suggested articles. On an e-commerce app, you'd be able to make and view your past orders, on Stack Overflow you can post questions and answers, whereas on a news site you might get a customized experience based on previous articles you've viewed.

When you think about adding users to your application, you typically have two aspects to consider:

- *Authentication*—The process of creating users and letting them log in to your app
- *Authorization*—Customizing the experience and controlling what users can do, based on the current logged-in user

In this chapter, I'm going to be discussing the first of these points, authentication and membership, and in the next chapter, I'll tackle the second point, authorization. In section 14.1, I discuss the difference between authentication and authorization, how authentication works in a traditional ASP.NET Core web app, and ways you can architect your system to provide sign-in functionality.

I also touch on the typical differences in authentication between a traditional web app and Web APIs used with client-side or mobile web apps. This book focuses on traditional web apps for authentication, but many of the principles are applicable to both.

In section 14.2, I introduce a user management system called ASP.NET Core Identity (or Identity for short). Identity integrates with EF Core and provides services for creating and managing users, storing and validating passwords, and signing users in and out of your app.

In section 14.3, you'll create an app using a default template that includes ASP.NET Core Identity out of the box. This will give you an app to explore, to see the features Identity provides, as well as everything it doesn't.

Creating an app is great for seeing an example of how the pieces fit together, but you'll often need to add users to an existing app. In section 14.4, you'll see each of the steps required to add ASP.NET Core Identity to an existing app: the recipe application from chapters 12 and 13.

In sections 14.5 and 14.6 you'll learn how to replace pages from the default Identity UI by "scaffolding" individual pages. In section 14.5 you'll see how to customize the Razor templates to generate different HTML on the user registration page, and in section 14.6 you'll learn how to customize the logic associated with a Razor Page. You'll see how to store additional information about a user (such as their name or date of birth) and to provide permissions to them that you can later use to customize the app's behavior (if the user is a VIP, for example).

Before we look at the ASP.NET Core Identity system specifically, let's take a look at authentication and authorization in ASP.NET Core, what's happening when you sign in to a website, and some of the ways to design your apps to provide this functionality.

14.1 Introducing authentication and authorization

When you add sign-in functionality to your app and control access to certain functions based on the currently-signed-in user, you're using two distinct aspects of security:

- *Authentication*—The process of determining *who you are*
- *Authorization*—The process of determining *what you're allowed to do*

Generally, you need to know *who* the user is before you can determine *what* they're allowed to do, so authentication always comes first, followed by authorization. In this chapter, we're only looking at authentication; we'll cover authorization in chapter 15.

In this section I start by discussing how ASP.NET Core thinks about users, and cover some of the terminology and concepts that are central to authentication. I always found this to be the hardest part to grasp when first learning about authentication, so I'll take it slow!

Next, we'll look at what it *means* to sign in to a traditional web app. After all, you only provide your password and sign in to an app on a single page—how does the app know the request came from *you* for subsequent requests?

Finally, we'll look at how authentication works when you need to support client-side apps and mobile apps that call Web APIs, in addition to traditional web apps. Many of the concepts are similar, but the requirement to support multiple types of users, traditional apps, client-side apps, and mobile apps has led to alternative solutions.

In the next section, we'll look at a practical implementation of a user management system called ASP.NET Core Identity, and how you can use this in your own projects.

14.1.1 Understanding users and claims in ASP.NET Core

The concept of a user is baked-in to ASP.NET Core. In this section, we'll look at the fundamental model used to describe a user, including its properties, such as an associated email address, as well as any permission-related properties, such as whether the user is an admin user.

In chapter 3, you learned that the HTTP server, Kestrel, creates an `HttpContext` object for every request it receives. This object is responsible for storing all the details related to that request, such as the request URL, any headers sent, the body of the request, and so on.

The `HttpContext` object also exposes the current *principal* for a request as the `User` property. This is ASP.NET Core's view of which user made the request. Any time your app needs to know who the current user is, or what they're allowed to do, it can look at the `HttpContext.User` principal.

> **DEFINITION** You can think of the *principal* as the user of your app.

In ASP.NET Core, principals are implemented as `ClaimsPrincipals`, which has a collection of *claims* associated with it, as shown in figure 14.1.

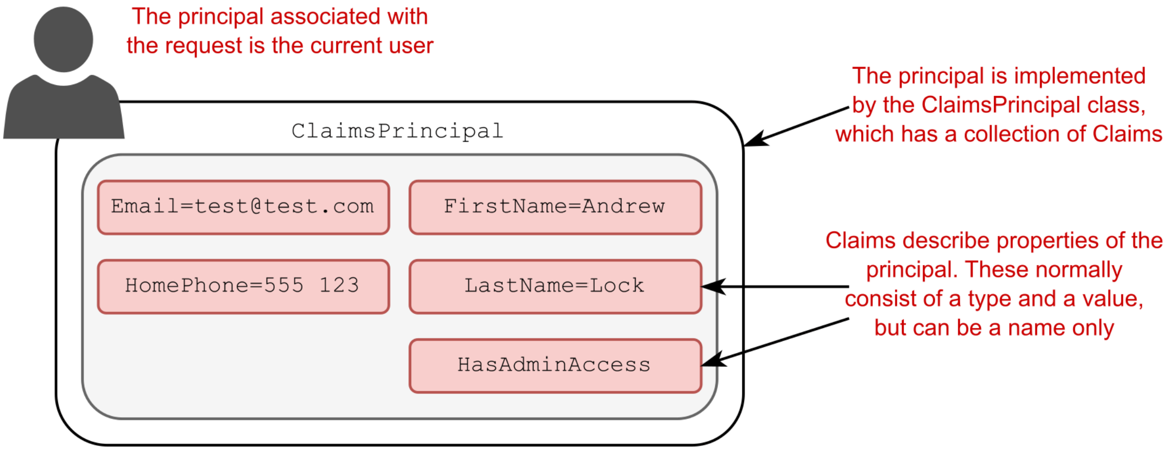

The principal associated with
the request is the current user
ClaimsPrincipal
Email=test@test.com
FirstName=Andrew
HomePhone=555 123
LastName=Lock
HasAdminAccess
The principal is implemented by the ClaimsPrincipal class, which has a collection of Claims
Claims describe properties of the principal. These normally consist of a type and a value, but can be a name only

Figure 14.1 The principal is the current user, implemented as `ClaimsPrincipal`. It contains a collection of `Claims` that describe the user.

You can think about claims as properties of the current user. For example, you could have claims for things like email, name, or date of birth.

> **DEFINITION** A claim is a single piece of information about a principal, which consists of a *claim type* and an optional *value*.

Claims can also be indirectly related to permissions and authorization, so you could have a claim called `HasAdminAccess` or `IsVipCustomer`. These would be stored in exactly the same way—as claims associated with the user principal.

> **NOTE** Earlier versions of ASP.NET used a role-based approach to security, rather than claims-based. The `ClaimsPrincipal` used in ASP.NET Core is compatible with this approach for legacy reasons, but you should use the claims-based approach for new apps.

Kestrel assigns a user principal to every request that arrives at your app. Initially, that principal is a generic, anonymous, unauthenticated principal with no claims. How do you log in, and how does ASP.NET Core know that you've logged in on subsequent requests?

In the next section, we'll look at how authentication works in a traditional web app using ASP.NET Core, and the process of signing in to a user account.

14.1.2 Authentication in ASP.NET Core: services and middleware

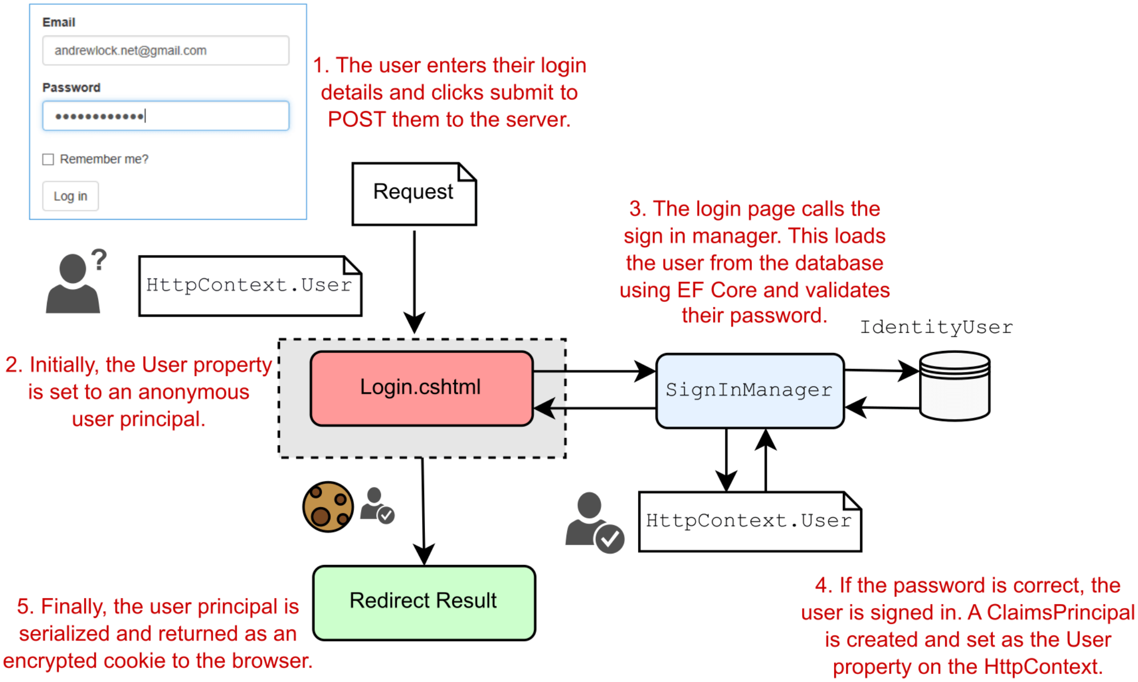
Adding authentication to any web app involves a number of moving parts. The same general process applies whether you're building a traditional web app or a client-side app, though there are often differences in implementation, as I'll discuss in section 14.1.3:

- The client sends an identifier and a secret to the app, which identify the current user. For example, you could send an email (identifier) and a password (secret).
- The app verifies that the identifier corresponds to a user known by the app and that the corresponding secret is correct.
- If the identifier and secret are valid, the app can set the principal for the current request, but it also needs a way of storing these details for subsequent requests. For traditional web apps, this is typically achieved by storing an encrypted version of the user principal in a cookie.

This is the typical flow for most web apps, but in this section, I'm going to look at how it works in ASP.NET Core. The overall process is the same, but it's good to see how this pattern fits into the services, middleware, and MVC aspects of an ASP.NET Core application. We'll step through the various pieces at play in a typical app when you sign in as a user, what that means, and how you can make subsequent requests as that user.

SIGNING IN TO AN ASP.NET CORE APPLICATION

When you first arrive on a site and sign in to a traditional web app, the app will send you to a sign-in page and ask you to enter your username and password. After you submit the form to the server, the app redirects you to a new page, and you're magically logged in! Figure 14.2 shows what's happening behind the scenes in an ASP.NET Core app when you submit the form.



Email

andrewlock.net@gmail.com

Password

•••••••••••••

☐ Remember me?

Log in

1. The user enters their login details and clicks submit to POST them to the server.

2. Initially, the User property is set to an anonymous user principal.

3. The login page calls the sign in manager. This loads the user from the database using EF Core and validates their password.

4. If the password is correct, the user is signed in. A ClaimsPrincipal is created and set as the User property on the HttpContext.

5. Finally, the user principal is serialized and returned as an encrypted cookie to the browser.

Figure 14.2 Signing in to an ASP.NET Core application. `SignInManager` is responsible for setting `HttpContext.User` to the new principal and serializing the principal to the encrypted cookie.

This shows the series of steps from the moment you submit the login form on a Razor Page, to the point the redirect is returned to the browser. When the request first arrives, Kestrel creates an anonymous user principal and assigns it to the `HttpContext.User` property. The request is then routed to the Login.chtml Razor Page, which reads the email and password from the request using model binding.

The meaty work happens inside the `SignInManager` service. This is responsible for loading a user entity with the provided username from the database and validating that the password they provided is correct.

WARNING Never store passwords in the database directly. They should be *hashed* using a strong one-way algorithm. The ASP.NET Core Identity system does this for you, but it's always wise to reiterate this point!

If the password is correct, `SignInManager` creates a new `ClaimsPrincipal` from the user entity it loaded from the database and adds the appropriate claims, such as the email. It then replaces the old, anonymous, `HttpContext.User` principal with the new, authenticated principal.

Finally, `SignInManager` serializes the principal, encrypts it, and stores it as a *cookie*. A cookie is a small piece of text that's sent back and forth between the browser and your app along with each request, consisting of a name and a value.

This authentication process explains how you can set the user for a request when they *first* log in to your app, but what about subsequent requests? You only send your password when you first log in to an app, so how does the app know that it's the same user making the request?

AUTHENTICATING USERS FOR SUBSEQUENT REQUESTS

The key to persisting your identity across multiple requests lies in the final step of figure 14.2, where you serialized the principal in a cookie. Browsers will automatically send this cookie with all requests made to your app, so you don't need to provide your password with every request.

ASP.NET Core uses the authentication cookie sent with the requests to rehydrate `ClaimsPrincipal` and set the `HttpContext.User` principal for the request, as shown in figure 14.3. The important thing to note is *when* this process happens—in the `AuthenticationMiddleware`.

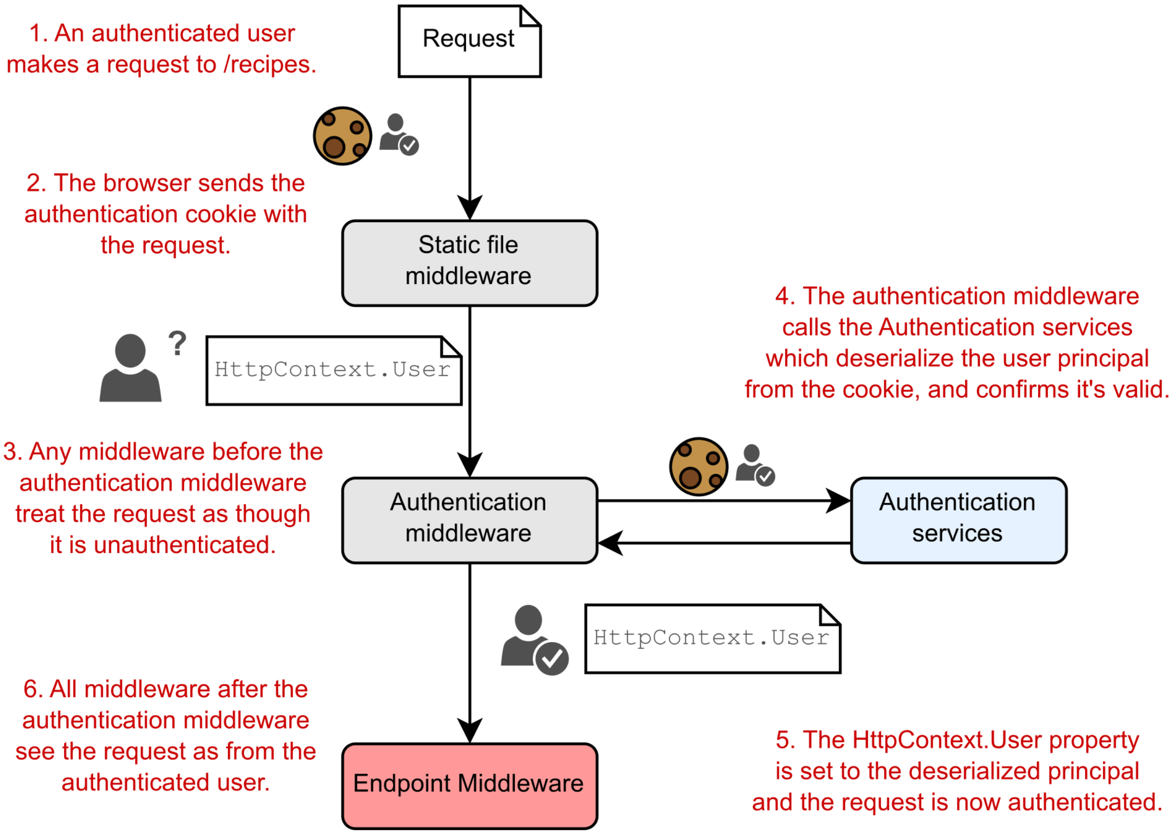

1. An authenticated user makes a request to /recipes.
Request
2. The browser sends the authentication cookie with the request.
Static file middleware
HttpContext.User
?
3. Any middleware before the authentication middleware treat the request as though it is unauthenticated.
4. The authentication middleware calls the Authentication services which deserialize the user principal from the cookie, and confirms it's valid.
Authentication middleware
Authentication services
HttpContext.User
5. The HttpContext.User property is set to the deserialized principal and the request is now authenticated.
6. All middleware after the authentication middleware see the request as from the authenticated user.
Endpoint Middleware

Figure 14.3 A subsequent request after signing in to an application. The cookie sent with the request contains the user principal, which is validated and used to authenticate the request.

When a request containing the authentication cookie is received, Kestrel creates the default, unauthenticated, anonymous principal and assigns it to the `HttpContext.User` principal. Any middleware that runs at this point, before `AuthenticationMiddleware`, will see the request as unauthenticated, even if there's a valid cookie.

> **TIP** If it looks like your authentication system isn't working, double-check your middleware pipeline. Only middleware that runs after `AuthenticationMiddleware` will see the request as authenticated.

`AuthenticationMiddleware` is responsible for setting the current user for a request. The middleware calls authentication services, which reads the cookie from the request, decrypts it, and deserializes it to obtain the `ClaimsPrincipal` created when the user logged in.

`AuthenticationMiddleware` sets the `HttpContext.User` principal to the new, authenticated principal. All subsequent middleware will now know the user principal for the request and can adjust behavior accordingly (for example, displaying the user's name on the homepage, or restricting access to some areas of the app).

> **NOTE The `AuthenticationMiddleware` is *only* responsible for authenticating incoming requests and setting the `ClaimsPrincipal` if the request contains an authentication cookie. It is *not* responsible for redirecting unauthenticated requests to the login page or rejecting unauthorized requests—that is handled by the `AuthorizationMiddleware`, as you'll see in chapter 15.**

The process described so far, in which a single app authenticates the user when they log in and sets a cookie that's read on subsequent requests, is common with traditional web apps, but it isn't the only possibility. In the next section, we take a look at authentication for Web API applications, used by client-side and mobile apps, and how the authentication system changes for those scenarios.

14.1.3 Authentication for APIs and distributed applications

The process I've outlined so far applies to traditional web apps, where you have a single endpoint that's doing all the work. It's responsible for authenticating and managing users, as well as serving your app data, as shown in figure 14.4.

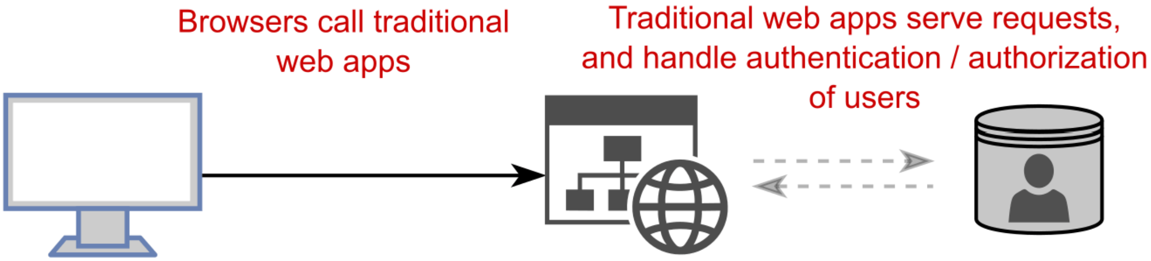

Browsers call traditional web apps
Traditional web apps serve requests, and handle authentication / authorization of users

Figure 14.4 Traditional apps typically handle all the functionality of an app: the business logic, generating the UI, authentication, and user management.

In addition to this traditional web app, it's common to use ASP.NET Core as a Web API to serve data for mobile and client-side SPAs. Similarly, the trend towards microservices on the backend means that even traditional web apps using Razor often need to call APIs behind the scenes, as shown in figure 14.5.

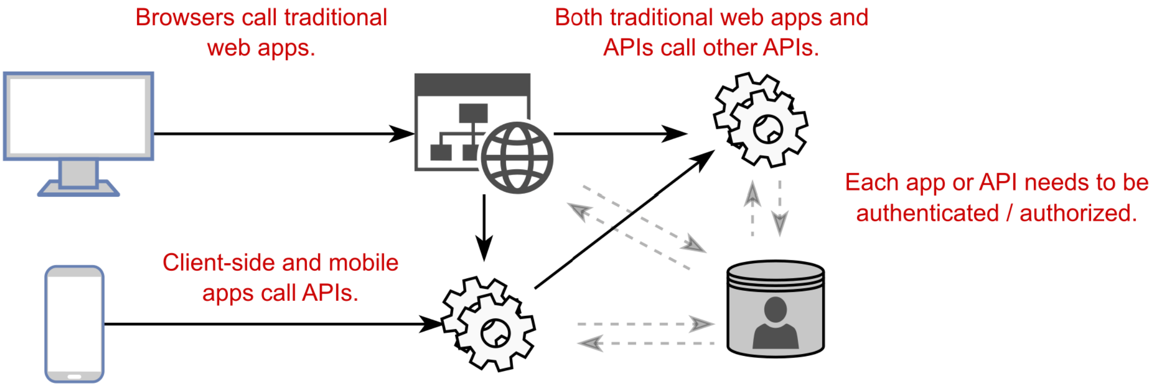

Browsers call traditional web apps.
Both traditional web apps and APIs call other APIs.
Each app or API needs to be authenticated / authorized.
Client-side and mobile apps call APIs.

Figure 14.5 Modern applications typically need to expose Web APIs for mobile and client-side apps, as well as potentially calling APIs on the backend. When all of these services need to authenticate and manage users, this becomes complicated logistically.

In this situation, you have multiple apps and APIs, all of which need to understand that the same user is making a request across all the apps and APIs. If you keep the same approach as before, where each app manages its own users, things can quickly become unmanageable!

You'd need to duplicate all the sign-in logic between the apps and APIs, as well as needing to have some central database holding the user details. Users may need to sign in multiple times to access different parts of the service. On top of that, using cookies becomes problematic for some mobile clients in particular, or where you're making requests to multiple domains (as cookies only belong to a single domain).

How can you improve this? The typical approach is to extract the code that's common to all of the apps and APIs, and move it to an *identity provider*, as shown in figure 14.6.

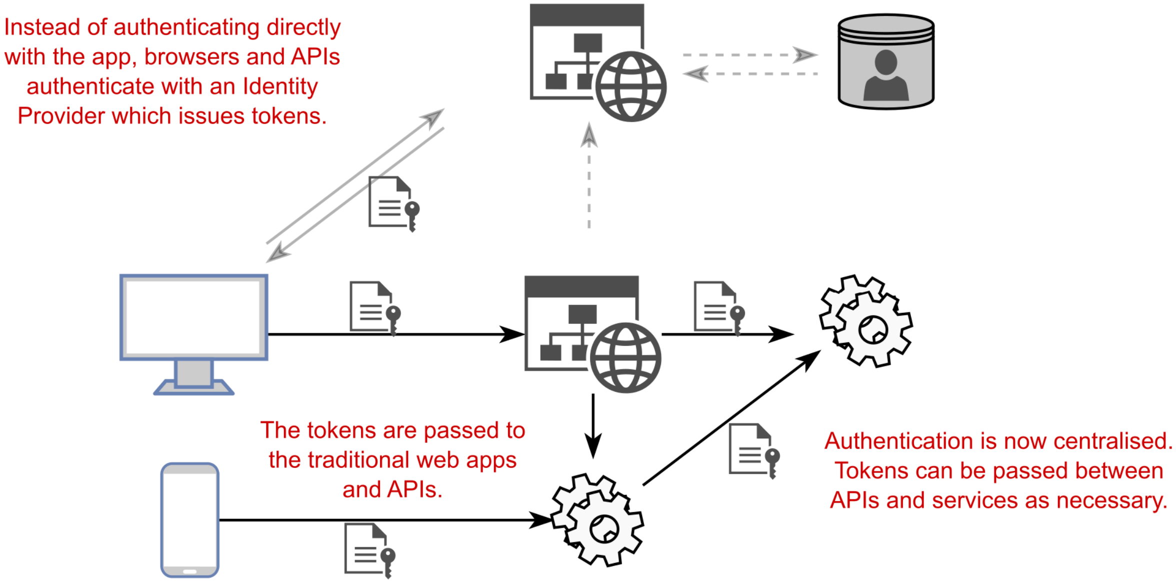

Instead of authenticating directly with the app, browsers and APIs authenticate with an Identity Provider which issues tokens.
The tokens are passed to the traditional web apps and APIs.
Authentication is now centralised. Tokens can be passed between APIs and services as necessary.

Figure 14.6 An alternative architecture involves using a central identity provider to handle all the authentication and user management for the system. Tokens are passed back and forth between the identity provider, apps, and APIs.

Instead of signing in to an app directly, the app redirects to an identity provider app. The user signs in to this identity provider, which passes *bearer tokens* back to the client that indicate who the user is and what they're allowed to access. The clients and apps can pass these tokens to the APIs, to provide information about the logged-in user, without needing to re-authenticate or manage users directly.

This architecture is clearly more complicated on the face of it, as you've thrown a whole new service—the identity provider—into the mix, but in the long run this has a number of advantages:

- *Users can share their identity between multiple services*. As you're logged in to the central identity provider, you're essentially logged in to *all* apps that use that service. This gives you the "single-sign-on" experience, where you don't have to keep logging in to multiple services.
- *Reduced duplication*. All of the sign-in logic is encapsulated in the identity provider, so you don't need to add sign-in screens to all your apps.
- *Can easily add new providers*. Whether you use the identity provider approach or the traditional approach, it's possible to use external services to handle the authentication of users. You'll have seen this on apps that allow you to "log in using Facebook" or "log in using Google," for example. If you use a centralized identity provider, adding support

for additional providers can be handled in one place, instead of having to configure every app and API explicitly.

Out of the box, ASP.NET Core supports architectures like this, and for *consuming* issued bearer tokens, but it doesn't include support for *issuing* those tokens in the core framework. That means you'll need to use another library or service for the identity provider.

One option for an identity provider is to delegate all the authentication responsibilities to a third-party identity provider, such as Facebook, Okta, Auth0, or Azure Active Directory B2C. These manage users for you, so user information and passwords are stored in their database, rather than your own. The biggest advantage of this approach is that you don't have to worry about making sure your customer data is safe; you can be pretty sure that a third party will protect it, as it's their whole business!

TIP Wherever possible, I recommend this approach, as it delegates security responsibilities to someone else. You can't lose your user's details if you never had them!

Another common option is to build your own identity provider. This may sound like a lot of work, but thanks to excellent libraries like OpenIddict (https://github.com/ openiddict) and IdentityServer4 (http://docs.identityserver.io), it's perfectly possible to write your own identity provider to serve bearer tokens that will be consumed by an application.

An aspect often overlooked by people getting started with OpenIddict and IdentityServer is that they aren't prefabricated solutions. You, as a developer, need to write the code that knows how to create a new user (normally in a database), how to load a user's details, and how to validate their password. In that respect, the development process of creating an identity provider is similar to the traditional web app with cookie authentication that I discussed in section 14.1.2.

In fact, you can almost think of an identity provider as a traditional web app that only has account management pages. It also has the ability to generate tokens for other services, but it contains no other app-specific logic. The need to manage users in a database, as well as providing an interface for users to log in, is common to both approaches and is the focus of this chapter.

NOTE Hooking up your apps and APIs to use an identity provider can require a fair amount of tedious configuration, both of the app and the identity provider. For simplicity, this book focuses on traditional web apps using the process outlined in section 14.1.2. ASP.NET Core includes a helper library for working with IdentityServer and client-side SPAs. For details on how to get started see the documentation at https://docs.microsoft.com/aspnet/core/security/authentication/identity-api-authorization and http://docs.identityserver.io.

ASP.NET Core Identity (hereafter shortened to Identity) is a system that makes building the user management aspect of your app (or identity provider app) simpler. It handles all of the boilerplate for saving and loading users to a database, as well as a number of best practices for security, such as user lock out, password hashing, and *two-factor authentication*.

DEFINITION *Two-factor authentication* (2FA) is where you require an extra piece of information to sign in, in addition to a password. This could involve sending a code to a user's phone by SMS, or using a mobile app to generate a code, for example.

In the next section, I'm going to talk about the ASP.NET Core Identity system, the problems it solves, when you'd want to use it, and when you might not want to use it. In section 14.3, we'll take a look at some code and see ASP.NET Core Identity in action.

14.2 What is ASP.NET Core Identity?

In this section I introduce ASP.NET Core Identity, describe the features it provides, and the features that you must provide yourself. I also address some of the arguments against ASP.NET Core Identity, and when you should and shouldn't consider using it.

Whether you're writing a traditional web app using Razor Pages or are setting up a new identity provider using a library like IdentityServer, you'll need a way of persisting details about your users, such as their usernames and passwords.

This might seem like a relatively simple requirement but, given this is related to security and people's personal details, it's important you get it right. As well as storing the claims for each user, it's important to store passwords using a strong hashing algorithm, to allow users to use 2FA where possible, and to protect against brute force attacks, to name a few of the many requirements!

Although it's perfectly possible to write all the code to do this manually and to build your own authentication and membership system, I highly recommend you don't.

I've already mentioned third-party identity providers such as Auth0 or Azure Active Directory B2C. These are Software-as-a-Service (SaaS) solutions that take care of the user management and authentication aspects of your app for you. If you're in the process of moving apps to the cloud generally, then solutions like these can make a lot of sense.

If you can't or don't want to use these third-party solutions, then I recommend you consider using the ASP.NET Core Identity system to store and manage user details in your database. ASP.NET Core Identity takes care of most of the boilerplate associated with authentication, but remains flexible and lets you control the login process for users if you need to.

NOTE ASP.NET *Core* Identity is an evolution of ASP.NET Identity, with some design improvements and converted to work with ASP.NET Core.

By default, ASP.NET Core Identity uses EF Core to store user details in the database. If you're already using EF Core in your project, then this is a perfect fit. Alternatively, it's possible to write your own stores for loading and saving user details in another way.

Identity takes care of the low-level parts of user management, as shown in table 14.1. As you can see from this list, Identity gives you a lot, but not everything—by a long shot!

Table 14.1 Which services are and aren't handled by ASP.NET Core Identity

Managed by ASP.NET Core Identity	Requires implementing by the developer
Database schema for storing users and claims.	UI for logging in, creating, and managing users (Razor Pages or controllers). This is included in an optional package, which provides a default UI.
Creating a user in the database.	Sending emails messages.
Password validation and rules.	Customizing claims for users (adding new claims).
Handling user account lockout (to prevent brute-force attacks).	Configuring third-party identity providers.
Managing and generating 2FA codes.	
Generating password-reset tokens.	
Saving additional claims to the database.	
Managing third-party identity providers (for example Facebook, Google, Twitter).	

The biggest missing piece is the fact that you need to provide all the UI for the application, as well as tying all the individual Identity services together to create a functioning sign-in process. That's a pretty big missing piece, but it makes the Identity system extremely flexible.

Luckily, ASP.NET Core includes a helper NuGet library, Microsoft.AspNetCore.Identity.UI, that gives you the whole of the UI boilerplate for free. That's over 30 Razor Pages, with functionality for logging in, registering users, using two-factor authentication, and using external login providers, among others! You can still customize these pages if you need to, but having a whole login process working out of the box, with no code required on your part is a huge win. We'll look at this library and how you use it in sections 14.3 and 14.4.

For that reason, I strongly recommend using the default UI as a starting point, whether you're creating an app or adding user management to an existing app. But the question still remains: when should you use Identity, and when should you consider rolling your own?

I'm a big fan of Identity, so I tend to suggest it in most situations, as it handles a lot of security-related things for you that are easy to mess up. I've heard several arguments against it, some of which are valid, and others less so:

- *I already have user authentication in my app*. Great! In that case, you're probably right, Identity may not be necessary. But does your custom implementation use 2FA? Do you have account lockout? If not, and you need to add them, then considering Identity may be worthwhile.

- *I don't want to use EF Core*. That's a reasonable stance. You could be using Dapper, some other ORM, or even a document database for your database access. Luckily, the database integration in Identity is pluggable, so you could swap out the EF Core integration and use your own database integration libraries instead.
- *My use case is too complex for Identity*. Identity provides lower-level services for authentication, so you can compose the pieces however you like. It's also extensible, so if you need to, for example, transform claims before creating a principal, you can.
- *I don't like the default Razor Pages UI*. The default UI for Identity is entirely optional. You can still use the Identity services and user management but provide your own UI for logging in and registering users. However, be aware that although doing this gives you a lot of flexibility, it's also very easy to introduce a security flaw in your user management system, the last place you want security flaws!
- *I'm not using Bootstrap to style my application*. The default Identity UI uses Bootstrap as a styling framework, the same as the default ASP.NET Core templates. Unfortunately, you can't easily change that, so if you're using a different framework, or need to customise the HTML generated, then you can still use Identity, but you'll need to provide your own UI.
- *I don't want to build my own identity system*. I'm glad to hear it. Using an external identity provider like Azure Active Directory B2C or Auth0 is a great way of shifting the responsibility and risk associated with storing users' personal information onto a third party.

Any time you're considering adding user management to your ASP.NET Core application, I'd recommend looking at Identity as a great option for doing so. In the next section, I'll demonstrate what Identity provides by creating a new Razor Pages application using the default Identity UI. In section 14.4, we'll take that template and apply it to an existing app instead, and in sections 14.5 and 14.6 you'll see how to override the default pages.

14.3 Creating a project that uses ASP.NET Core Identity

I've covered authentication and Identity in general terms a fair amount now, but the best way to get a feel for it is to see some working code. In this section, we're going to look at the default code generated by the ASP.NET Core templates with Identity, how the project works, and where Identity fits in.

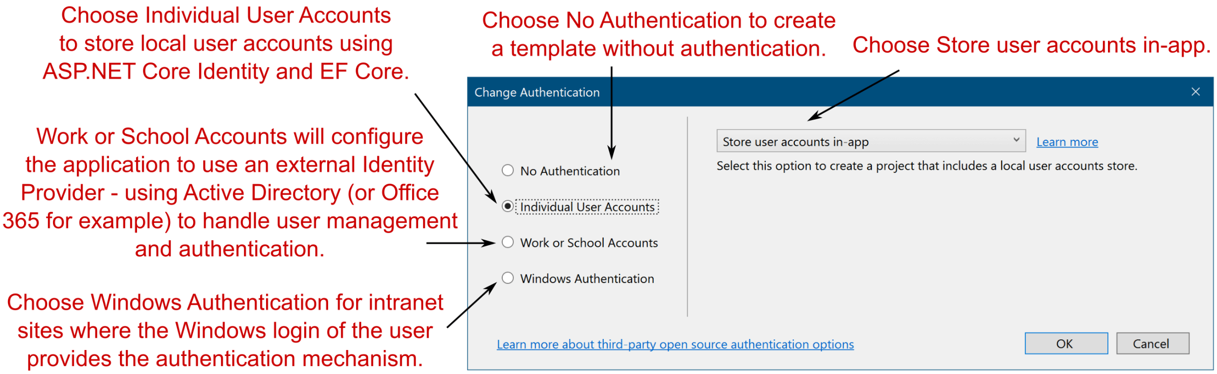
14.3.1 Creating the project from a template

You'll start by using the Visual Studio templates to generate a simple Razor Pages application that uses Identity for storing individual user accounts in a database.

> **TIP** You can create an equivalent project using the .NET CLI by running `dotnet new webapp -au Individual -uld`

To create the template using Visual Studio, you must be using VS 2019 or later and have the ASP.NET Core 3.1 SDK installed:

1. Choose File > New > Project or choose Create a New Project from the splash screen
2. From the list of templates, choose ASP.NET Core Web Application, ensuring you select the C# language template.
3. On the next screen, enter a project name, Location, and a solution name, and click Create.
4. Choose the Web Application template and click Change under Authentication to bring up the Authentication dialog, shown in figure 14.7.



Change Authentication

○ No Authentication
● Individual User Accounts
○ Work or School Accounts
○ Windows Authentication

Store user accounts in-app

Select this option to create a project that includes a local user accounts store.

Learn more
Learn more about third-party open source authentication options
OK
Cancel

Choose Individual User Accounts to store local user accounts using ASP.NET Core Identity and EF Core.

Choose No Authentication to create a template without authentication.

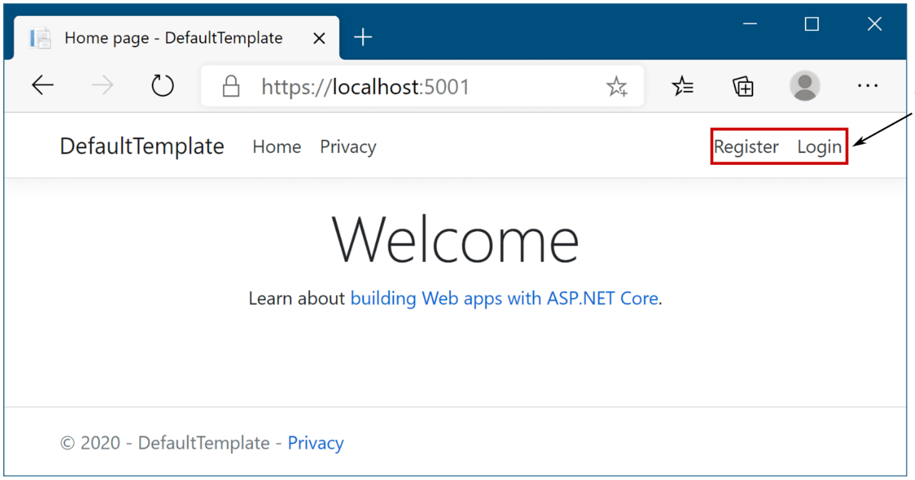
Choose Store user accounts in-app.

Work or School Accounts will configure the application to use an external Identity Provider - using Active Directory (or Office 365 for example) to handle user management and authentication.

Choose Windows Authentication for intranet sites where the Windows login of the user provides the authentication mechanism.

Figure 14.7 Choosing the authentication mode of the new ASP.NET Core application template in VS 2019

5. Choose Individual User Accounts to create an application configured with EF Core and ASP.NET Core Identity. Click OK.
6. Click Create to create the application. Visual Studio will automatically run `dotnet restore` to restore all the necessary NuGet packages for the project.
7. Run the application to see the default app, as shown in figure 14.8.


Home page - DefaultTemplate
https://localhost:5001
DefaultTemplate
Home
Privacy
Register
Login
Welcome
Learn about building Web apps with ASP.NET Core.
© 2020 - DefaultTemplate - Privacy

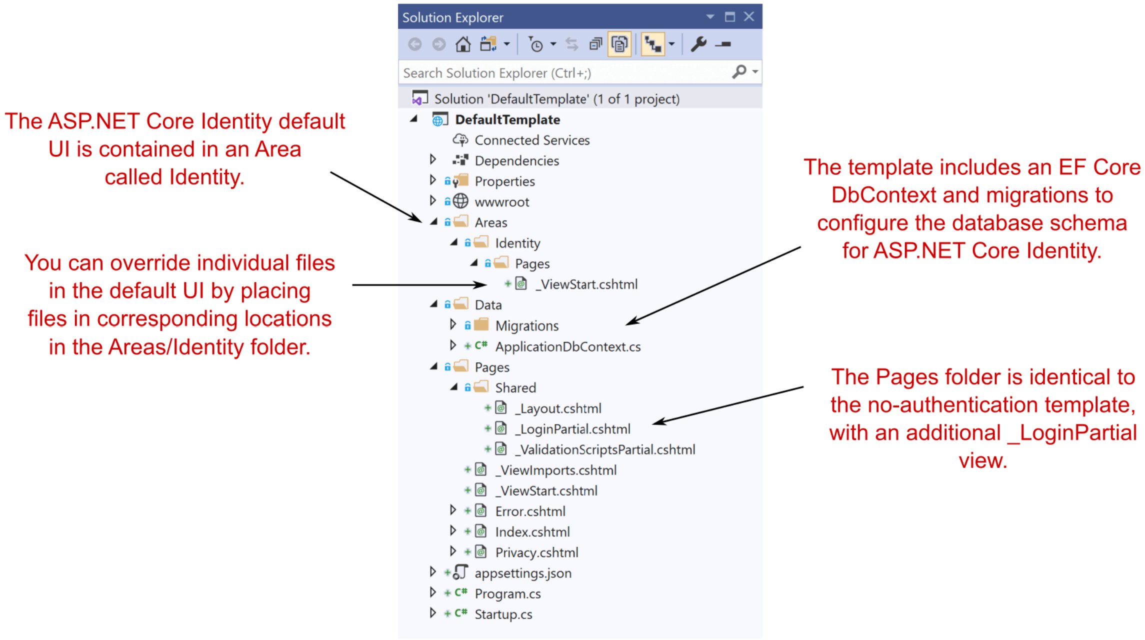
You can create new users and sign in using the log in widget.

Figure 14.8 The default template with individual account authentication looks similar to the no authentication template, with the addition of a login widget in the top right of the page.

This template should look familiar, with one twist: you now have Register and Login buttons! Feel free to play with the template—creating a user, logging in and out—to get a feel for the app. Once you're happy, look at the code generated by the template and the boilerplate it saved you from writing!

14.3.2 Exploring the template in Solution Explorer

The project generated by the template, shown in figure 14.9, is very similar to the default no-authentication template. That's largely due to the default UI library which brings in a big chunk of functionality without exposing you to the nitty gritty details.



The ASP.NET Core Identity default UI is contained in an Area called Identity.

You can override individual files in the default UI by placing files in corresponding locations in the Areas/Identity folder.

The template includes an EF Core DbContext and migrations to configure the database schema for ASP.NET Core Identity.

The Pages folder is identical to the no-authentication template, with an additional _LoginPartial view.

Figure 14.9 The project layout of the default template. Depending on your version of Visual Studio, the exact files may vary slightly.

The biggest addition is the Areas folder in the root of your project which contains an Identity subfolder. Areas are sometimes used for organizing sections of functionality. Each area can contain its own Pages folder, which is analogous to the main Pages folder in your application.

> **DEFINITION** ***Areas*** **are used to group Razor Pages into separate hierarchies for organizational purposes. I rarely use areas and prefer to create sub folders in the main Pages folder instead. The one exception is the default Identity UI which uses a separate Identity area by default. For more details on areas, see https://docs.microsoft.com/aspnet/core/mvc/controllers/areas.**

The Microsoft.AspNetCore.Identity.UI package creates Razor Pages in the "Identity" area. You can override any page in this default UI by creating a corresponding page in the Areas/Identity/Pages folder in your application. For example, as shown in figure 14.9, the default template adds a _ViewStart.cshtml file that overrides the version that is included as part of the default UI. This file contains the following code, which sets the default Identity UI Razor Pages to use your project's default _Layout.cshtml file:

```
@{
    Layout = "/Pages/Shared/_Layout.cshtml";
}
```

Some obvious questions at this point might be "how do you know what's included in the default UI", and "which files you can override"? You'll see the answer to both of those in section 14.5, but in general you should try and avoid overriding files where possible. After all, the goal with the default UI is to *reduce* the amount of code you have to write!

The Data folder in your new project template contains your application's EF Core `DbContext`, called `ApplicationDbContext`, and the migrations for configuring the database schema to use Identity. I'll discuss this schema in more detail in section 14.3.3.

The final additional file included in this template compared to the no-authentication version is the partial Razor view Pages/Shared/_LoginPartial.cshtml. This provides the Register and Login links you saw in figure 14.8, and is rendered in the default Razor layout, _Layout.cshtml.

If you look inside _LoginPartial.cshtml, you can see how routing works with areas, by combining the Razor Page path with an `{area}` route parameter using Tag Helpers. For example, the Login link specifies that the Razor Page /Account/Login is in the `Identity` area using the asp-area attribute:

```
<a asp-area="Identity" asp-page="/Account/Login">Login</a>
```

TIP You can reference Razor Pages in the `Identity` area by setting the `area` route value to `Identity`. You can use the `asp-area` attribute in Tag Helpers that generate links.

In addition to the new files included thanks to ASP.NET Core Identity, it's worth opening up Startup.cs and looking at the changes there. The most obvious change is the additional configuration in `ConfigureServices`, which adds all the services Identity requires.

Listing 14.1 Adding ASP.NET Core Identity services to `ConfigureServices`

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>              #A
        options.UseSqlServer(                                           #A
            Configuration.GetConnectionString("DefaultConnection")));   #A
    services.AddDefaultIdentity<IdentityUser>(options =>                #B
            options.SignIn.RequireConfirmedAccount = true)              #C
          .AddEntityFrameworkStores<ApplicationDbContext>();               #D
    services.AddRazorPages();
}
```

#A ASP.NET Core Identity uses EF Core, so it includes the standard EF Core configuration.
#B Adds the Identity system, including the default UI, and configures the user type as IdentityUser
#C Require users to confirm their accounts (typically by email) before they log in
#D Configures Identity to store its data in EF Core

The `AddDefaultIdentity()` extension method does several things:

- Adds the core ASP.NET Core Identity services.

- Configures the application user type to be IdentityUser. This is the entity model that is stored in the database and represents a "user" in your application. You can extend this type if you need to, but that's not always necessary, as you'll see in section 14.6.
- Adds the default UI Razor Pages for registering, logging in, and managing users.
- Configures token providers for generating 2FA and email confirmation tokens.

There's another, very important, change in Startup, in the Configure method:

```
app.UseAuthentication();
```

This adds AuthenticationMiddleware to the pipeline, so that you can authenticate incoming requests, as you saw in figure 14.3. *The location of this middleware is very important*. It should be placed after UseRouting(), and before UseAuthorization() and UseEndpoints() as shown in the following listing.

Listing 14.2 Adding AuthenticationMiddleware to your middleware pipeline

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseStaticFiles();                #A

    app.UseRouting();                    #B

    app.UseAuthentication();             #C
    app.UseAuthorization();              #D

    app.UseEndpoints(endpoints =>        #E
    {                                    #E
        endpoints.MapRazorPages();       #E
    });                                  #E
}
```

#A Middleware placed before UseAuthentication will see all requests as anonymous
#B The routing middleware determines which page is requested based on the request URL
#C UseAuthentication should be placed after UseRouting
#D UseAuthorization should be placed after UseAuthentication, so it can access the user principal
#E UseEndpoints should be last, after the user principal is set and authorization has been applied

If you don't use this specific order of middleware, you can run into strange bugs where users aren't authenticated correctly, or authorization policies aren't correctly applied. This order is configured for you automatically in your templates, but it's something to be careful about if you're upgrading an existing application, or moving middleware around.

> **IMPORTANT** UseAuthentication() **and** UseAuthorization() **must be placed between** UseRouting() **and** UseEndpoints()**. Additionally,** UseAuthorization() **must be placed after** UseAuthentication()**. You can add additional middleware between each of these calls, as long as this overall middleware order is preserved.**

Now you've got an overview of the additions made by Identity, we'll look in a bit more detail at the database schema and how Identity stores users in the database.

14.3.3 The ASP.NET Core Identity data model

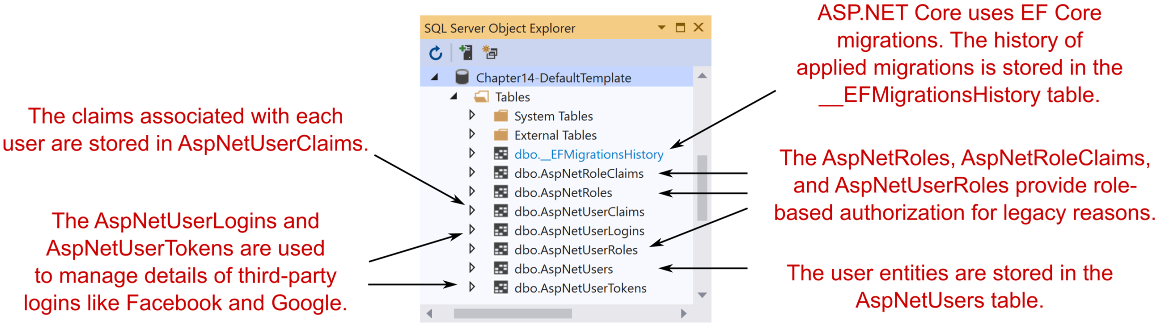
Out of the box, and in the default templates, Identity uses EF Core to store user accounts. It provides a base `DbContext` that you can inherit from, called `IdentityDbContext`, which uses an `IdentityUser` as the user entity for your application.

In the template, the app's `DbContext` is called `ApplicationDbContext`. If you open up this file, you'll see it's very sparse; it inherits from the `IdentityDbContext` base class I described earlier, and that's it. What does this base class give you? The easiest way to see is to update a database with the migrations and take a look!

Applying the migrations is the same process as in chapter 12. Ensure the connection string points to where you want to create the database, open a command prompt in your project folder, and run

```
dotnet ef database update
```

to update the database with the migrations. If the database doesn't yet exist, the CLI will create it. Figure 14.10 shows what the database looks like for the default template.[63]



The claims associated with each user are stored in AspNetUserClaims.

The AspNetUserLogins and AspNetUserTokens are used to manage details of third-party logins like Facebook and Google.

ASP.NET Core uses EF Core migrations. The history of applied migrations is stored in the __EFMigrationsHistory table.

The AspNetRoles, AspNetRoleClaims, and AspNetUserRoles provide role-based authorization for legacy reasons.

The user entities are stored in the AspNetUsers table.

Figure 14.10 The database schema used by ASP.NET Core Identity

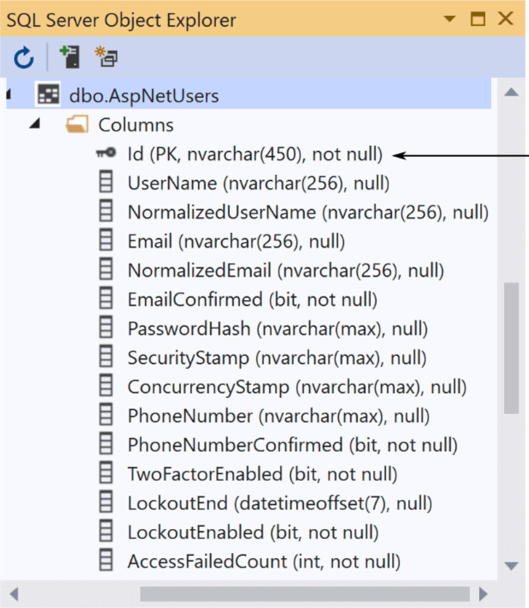
That's a lot of tables! You shouldn't need to interact with these tables directly—Identity handles that for you—but it doesn't hurt to have a basic grasp of what they're for:

- *__EFMigrationsHistory*—The standard EF Core migrations table that records which migrations have been applied.
- *AspNetUsers*—The user profile table itself. This is where `IdentityUser` is serialized to. We'll take a closer look at this table shortly.

[63] If you're using MS SQL Server (or LocalDB), you can use the SQL Server Object Explorer in Visual Studio to browse tables and objects in your database. See https://docs.microsoft.com/sql/ssdt/how-to-connect-to-a-database-and-browse-existing-objects for details.

- *AspNetUserClaims*—The claims associated with a given user. A user can have many claims, so it's modeled as a many-to-one relationship.
- *AspNetUserLogins and AspNetUserTokens*—These are related to third-party logins. When configured, these let users sign in with a Google or Facebook account (for example), instead of creating a password on your app.
- *AspNetUserRoles, AspNetRoles, and AspNetRoleClaims*—These tables are somewhat of a legacy left over from the old role-based permission model of the pre-.NET 4.5 days, instead of the claims-based permission model. These tables let you define roles that multiple users can belong to. Each role can be assigned a number of claims. These claims are effectively inherited by a user principal when they are assigned that role.

You can explore these tables yourself, but the most interesting of them is the AspNetUsers table, shown in figure 14.11.



```
dbo.AspNetUsers
  Columns
    Id (PK, nvarchar(450), not null)
    UserName (nvarchar(256), null)
    NormalizedUserName (nvarchar(256), null)
    Email (nvarchar(256), null)
    NormalizedEmail (nvarchar(256), null)
    EmailConfirmed (bit, not null)
    PasswordHash (nvarchar(max), null)
    SecurityStamp (nvarchar(max), null)
    ConcurrencyStamp (nvarchar(max), null)
    PhoneNumber (nvarchar(max), null)
    PhoneNumberConfirmed (bit, not null)
    TwoFactorEnabled (bit, not null)
    LockoutEnd (datetimeoffset(7), null)
    LockoutEnabled (bit, not null)
    AccessFailedCount (int, not null)
```

By default, ASP.NET Core Identity uses Guids for the user Id, stored as strings in the database.

The table contains all the neccessary fields for authenticating a user, email and phone number confirmation, two factor authentication, and account lockout.

Figure 14.11 The AspNetUsers table is used to store all the details required to authenticate a user.

Most of the columns in the AspNetUsers table are security related—the user's email, password hash, whether they have confirmed their email, whether they have 2FA enabled, and so on. By default, there are no columns for additional information, like the user's name, for example.

> **NOTE** You can see from figure 14.11 that the primary key `Id` is stored as a string column. By default, Identity uses `Guid` for the identifier. To customize the data type, see

https://docs.microsoft.com/aspnet/core/security/authentication/customize-identity-model#change-the-primary-key-type.

Any additional properties of the user are stored as claims in the AspNetUserClaims table associated with that user. This lets you add arbitrary additional information, without having to change the database schema to accommodate it. Want to store the user's date of birth? You could add a claim to that user—no need to change the database schema. You can see this in action in section 14.6, when you add a Name claim to every new user.

> **NOTE** Adding claims is often the easiest way to extend the default `IdentityUser`, but you can also add additional properties to the `IdentityUser` directly. This requires database changes but is nevertheless useful in many situations. You can read how to add custom data using this approach here: https://docs.microsoft.com/aspnet/core/security/authentication/add-user-data.

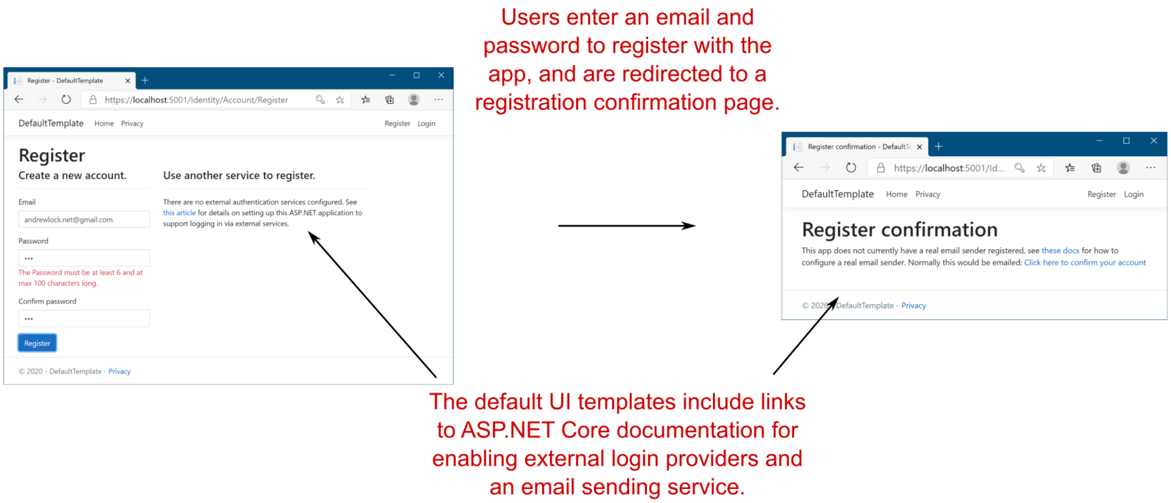
It's important to understand the difference between the `IdentityUser` entity (stored in the AspNetUsers table) and the `ClaimsPrincipal` which is exposed on `HttpContext.User`. When a user first logs in, an `IdentityUser` is loaded from the database. This entity is combined with additional claims for the user from the AspNetUserClaims table to create a `ClaimsPrincipal`. It's this `ClaimsPrincipal` which is used for authentication, and is serialized to the authentication cookie, not the `IdentityUser`.

It's useful to have a mental model of the underlying database schema Identity uses, but in day-to-day work, you shouldn't have to interact with it directly—that's what Identity is for after all! In the next section, we'll look at the other end of the scale—the UI of the app, and what you get out-of-the box with the default UI.

14.3.4 Interacting with ASP.NET Core Identity

You'll want to explore the default UI yourself, to get a feel for how the pieces fit together, but in this section, I'll highlight what you get out of the box, as well as areas that typically require additional attention right away.

The entry point to the default UI is the user registration page of the application, shown in figure 14.12. The register page enables users to sign up to your application by creating a new `IdentityUser` with an email and a password. After creating an account, users are redirected to a screen indicating they should confirm their email. No email service is enabled by default, as this is dependent on you configuring an external email service. You can read how to enable email sending at https://docs.microsoft.com/aspnet/core/security/authentication/accconfirm. Once you configure this, users will automatically receive an email with a link to confirm their account.


Register - DefaultTemplate
https://localhost:5001/Identity/Account/Register
DefaultTemplate
Home
Privacy
Register
Login

Register

Create a new account.

Email

andrewlock.net@gmail.com

Password

•••

The Password must be at least 6 and at max 100 characters long.

Confirm password

•••

Register

Use another service to register.

There are no external authentication services configured. See this article for details on setting up this ASP.NET application to support logging in via external services.

© 2020 - DefaultTemplate - Privacy

Register confirmation - DefaultT
https://localhost:5001/Id...
DefaultTemplate
Home
Privacy
Register
Login

Register confirmation

This app does not currently have a real email sender registered, see these docs for how to configure a real email sender. Normally this would be emailed: Click here to confirm your account

© 2020 - DefaultTemplate - Privacy

Users enter an email and password to register with the app, and are redirected to a registration confirmation page.

The default UI templates include links to ASP.NET Core documentation for enabling external login providers and an email sending service.

Figure 14.12 The registration flow for users using the default Identity UI. Users enter an email and password and are redirected to a "confirm your email page". This is a placeholder page by default, but if you enable email confirmation, this page will update appropriately.

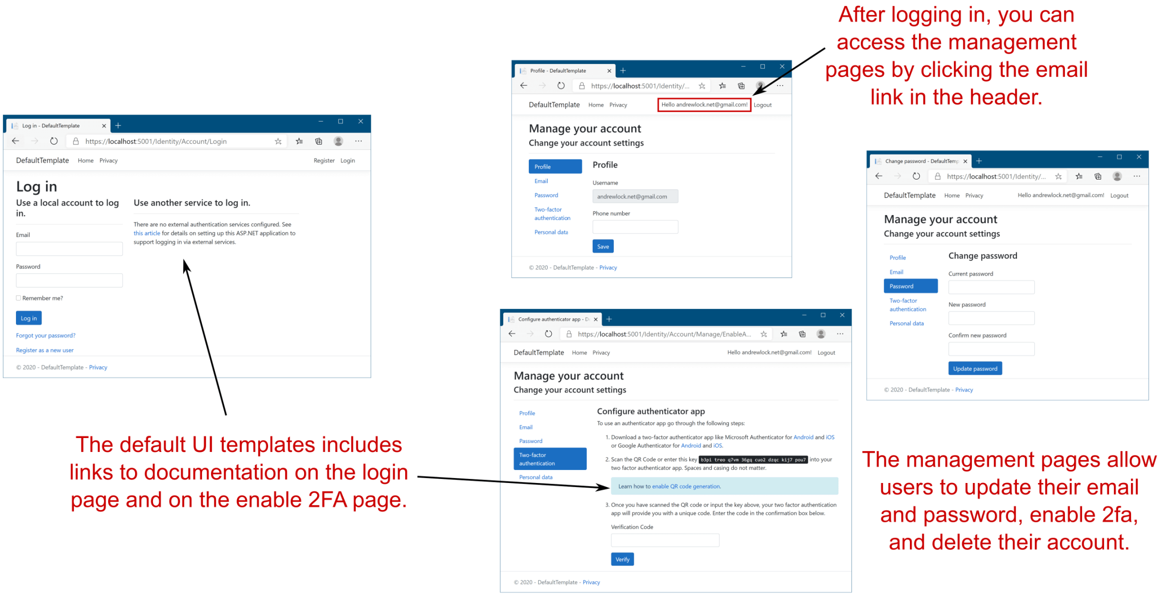
By default, user emails must be unique (you can't have two users with the same email) and the password must meet various length and complexity requirements. You can customize these options and more in the configuration lambda of the call to `AddDefaultIdentity()` in Startup.cs, as shown in the following listing.

Listing 14.3 Customizing Identity settings in ConfigurationService in Startup.cs

```
services.AddDefaultIdentity<ApplicationUser>(options =>
{
    options.SignIn.RequireConfirmedAccount = true;        #A
    options.Lockout.AllowedForNewUsers = true;            #B
    options.Password.RequiredLength = 12;                 #C
    options.Password.RequireNonAlphanumeric = false;      #C
    options.Password.RequireDigit = false;                #C
})
.AddEntityFrameworkStores<AppDbContext>();
```

#A Require users to confirm their account by email before they can log in
#B Enables user-lockout, to prevent brute-force attacks against user passwords
#C Update password requirements. Current guidance is to require long passwords.

After a user has registered with your application, they need to log in, as shown in figure 14.13. On the right-hand side of the login page, the default UI templates describe how you, the developer, can configure external login providers, such as Facebook and Google. This is useful information for you, but is one of the reasons you may need to customize the default UI templates, as you'll see in section 14.5.


Log in - DefaultTemplate
https://localhost:5001/Identity/Account/Login
DefaultTemplate
Home
Privacy
Register
Login
Log in
Use a local account to log in.
Email
Password
□ Remember me?
Log in
Forgot your password?
Register as a new user
© 2020 - DefaultTemplate - Privacy
Profile - DefaultTemplate
https://localhost:5001/Identity/...
Hello andrewlock.net@gmail.com!
Logout
Manage your account
Change your account settings
Profile
Email
Password
Two-factor authentication
Personal data
Username
Phone number
Save
Change password - DefaultTemp
Change password
Current password
New password
Confirm new password
Update password
Configure authenticator app - D
https://localhost:5001/Identity/Account/Manage/EnableA...
Configure authenticator app
Verification Code
Verify

Use another service to log in.

There are no external authentication services configured. See this article for details on setting up this ASP.NET application to support logging in via external services.

Username: andrewlock.net@gmail.com

To use an authenticator app go through the following steps:

1. Download a two-factor authenticator app like Microsoft Authenticator for Android and iOS or Google Authenticator for Android and iOS.
2. Scan the QR Code or enter this key `b3pi treo q7vm 36gq cuo2 dzqc kij7 pou7` into your two factor authenticator app. Spaces and casing do not matter.

   Learn how to enable QR code generation.
3. Once you have scanned the QR code or input the key above, your two factor authentication app will provide you with a unique code. Enter the code in the confirmation box below.

After logging in, you can access the management pages by clicking the email link in the header.

The default UI templates includes links to documentation on the login page and on the enable 2FA page.

The management pages allow users to update their email and password, enable 2fa, and delete their account.

Figure 14.13. Logging in with an existing user and managing the user account. The Login page describes how to configure external login providers, such as Facebook and Google. The user management pages allow users to change their email and password, and to configure Two Factor Authentication (2FA).

Once a user has signed in, they can access the management pages of the identity UI. These allow users to change their email, change their password, configure 2FA with an authenticator app, or delete all their personal data. Most of these functions work without any effort on your part, assuming you've already configured an email sending service.[64]

That covers everything you get in the default UI templates. It may seem somewhat minimal, but it covers a lot of the requirements that are common to almost all apps. Nevertheless, there are a few things you'll nearly always want to customize:

- Configure an email sending service, to enable account confirmation and password recovery, as described in the documentation: https://docs.microsoft.com/aspnet/core/security/authentication/accconfirm.
- Add a QR code generator for the enable 2FA page, as described in the documentation: https://docs.microsoft.com/aspnet/core/security/authentication/identity-enable-qrcodes.

[64] You can improve the 2FA authenticator page by enabling QR code generation, as described in this document: https://docs.microsoft.com/aspnet/core/security/authentication/identity-enable-qrcodes.

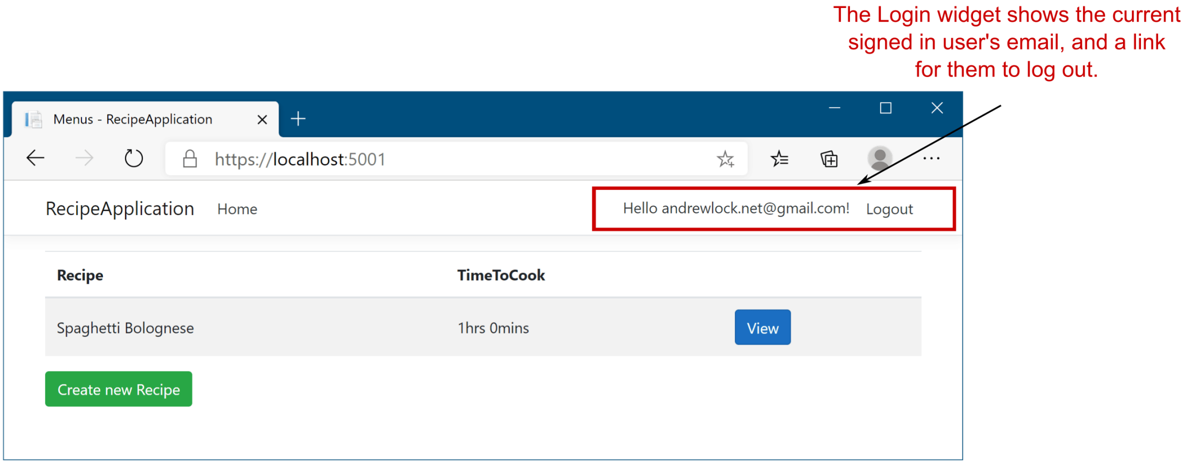
- Customise the register and log in pages to remove the documentation link for enabling external services. You'll see how to do this in section 14.5. Alternatively, you may want to disable user registration entirely, as described in the documentation: https://docs.microsoft.com/aspnet/core/security/authentication/scaffold-identity#disable-register-page.
- Collect additional information about users on the registration page. You'll see how to do this in section 14.6.

There are many more ways you can extend or update the Identity system, and lots of options available, so I encourage you to explore the documentation at https://docs.microsoft.com/aspnet/core/security/authentication to see your options. In the next section, you'll see how to achieve another common requirement: adding users to an existing application.

14.4 Adding ASP.NET Core Identity to an existing project

In this section, we're going to add users to the recipe application from chapters 12 and 13. This is a working app that you want to add user functionality to. In chapter 15, we'll extend this work to restrict control regarding who's allowed to edit recipes on the app.

By the end of this section, you'll have an application with a registration page, a login screen, and a manage account screen, like the default templates. You'll also have a persistent widget in the top right of the screen, showing the login status of the current user, as shown in figure 14.14.


The Login widget shows the current signed in user's email, and a link for them to log out.

Recipe	TimeToCook	
Spaghetti Bolognese	1hrs 0mins	View

Figure 14.14 The recipe app after adding authentication, showing the login widget.

As in section 14.3, I'm not going to customize any of the defaults at this point, so we won't set up external login providers, email confirmation, or 2FA; I'm only concerned with adding ASP.NET Core Identity to an existing app that's already using EF Core.

> **TIP** It's worth making sure you're comfortable with the new project templates before you go about adding Identity to an existing project. Create a test app and consider setting up an external login provider, configuring an email provider, and enabling 2FA. This will take a bit of time, but it'll be invaluable for deciphering errors when you come to adding Identity to existing apps.

To add Identity to your app, you'll need to:

1. Add the ASP.NET Core Identity NuGet packages.
2. Configure `Startup` to use `AuthenticationMiddleware` and add Identity services to the DI container.
3. Update the EF Core data model with the Identity entities.
4. Update your Razor Pages and layouts to provide links to the Identity UI.

This section will tackle each of these steps in turn. At the end of section 14.4, you'll have successfully added user accounts to the recipe app.

14.4.1 Configuring the ASP.NET Core Identity services and middleware

You can add ASP.NET Core Identity with the default UI to an existing app by referencing two NuGet packages:

- Microsoft.AspNetCore.Identity.EntityFrameworkCore—Provides all the core Identity services and integration with EF Core.
- Microsoft.AspNetCore.Identity.UI—Provides the default UI Razor Pages.

Update your project .csproj to include these two packages:

```
<PackageReference
    Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore"
    Version="3.1.3" />
<PackageReference
    Include="Microsoft.AspNetCore.Identity.UI" Version="3.1.3" />
```

These packages bring in all the additional required dependencies you need to add Identity with the default UI. Be sure to run `dotnet restore` after adding them to your project.

Once you've added the Identity packages, you can update your Startup.cs file to include the Identity services, as shown next. This is similar to the default template setup you saw in listing 14.1, but make sure to reference your existing `AppDbContext`.

Listing 14.4 Adding ASP.NET Core Identity services to the recipe app

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options =>                      #A
        options.UseSqlServer(                                           #A
            Configuration.GetConnectionString("DefaultConnection")));    #A

    services.AddDefaultIdentity<ApplicationUser>(options =>              #B
            options.SignIn.RequireConfirmedAccount = true)               #B
        .AddEntityFrameworkStores<AppDbContext>();                       #C

    services.AddRazorPages();
    services.AddScoped<RecipeService>();
}
```

#A The existing service configuration is unchanged.
#B Adds the Identity services to the DI container. Uses a custom user type, ApplicationUser
#C Makes sure you use the name of your existing DbContext app

This adds all the necessary services and configures Identity to use EF Core. I've introduced a new type here `ApplicationUser`, which we'll use to customize our user entity later. You'll see how to add this type in section 14.4.2.

Configuring `AuthenticationMiddleware` is somewhat easier: add it to the pipeline in the `Configure` method. As you can see in listing 14.7, I've added the middleware after `UseRouting()`, just before `UseAuthorization()`. As I mentioned in section 14.3.2, it's important you use this order for middleware in your application.

Listing 14.5 Adding `AuthenticationMiddleware` to the recipe app

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // other configuration not shown
    app.UseStaticFiles();             #A

    app.UseRouting();

    app.UseAuthentication();          #B
    app.UseAuthorization();           #C

    app.UseEndpoints(endpoints =>     #C
    {                                 #C
        endpoints.MapRazorPages();    #C
    });                               #C
}
```

#A StaticFileMiddleware will never see requests as authenticated, even after you sign in.
#B Adds AuthenticationMiddleware, after UseRouting() and before UseAuthorization
#C Middleware after AuthenticationMiddleware can read the user principal from HttpContext.User.

You've configured your app to use Identity, so the next step is to update EF Core's data model. You're already using EF Core in this app, so you need to update your database schema to include the tables that Identity requires.

14.4.2 Updating the EF Core data model to support Identity

The code in listing 14.4 won't compile, as it references the `ApplicationUser` type, which doesn't yet exist. Create the `ApplicationUser` in the Data folder, using the following:

```
public class ApplicationUser : IdentityUser { }
```

It’s not strictly necessary to create a custom user type (for example the default templates use the raw IdentityUser) but I find it’s easier to add now rather than try and retrofit it later if you need to customize the type.

In section 14.3.3 you saw that Identity provides a DbContext called IdentityDbContext, which you can inherit from. The IdentityDbContext base class includes the necessary DbSet<T> to store your user entities using EF Core.

Updating an existing DbContext for Identity is simple—update your app’s DbContext to inherit from IdentityDbContext, as shown in the following listing. We’re using the generic version of the base Identity context in this case and providing the ApplicationUser type.

Listing 14.6 Updating AppDbContext to use IdentityDbContext

```
public class AppDbContext : IdentityDbContext<ApplicationUser>     #A
{
    public AppDbContext(DbContextOptions<AppDbContext> options)    #B
        : base(options)                                            #B
    { }                                                            #B
                                                                   #B
    public DbSet<Recipe> Recipes { get; set; }                     #B
}
```

#A Updates to inherit from the Identity context, instead of directly from DbContext
#B The remainder of the class remains the same

Effectively, by updating the base class of your context in this way, you’ve added a whole load of new entities to EF Core’s data model. As you saw in chapter 12, whenever EF Core’s data model changes, you need to create a new migration and apply those changes to the database.

At this point, your app should compile, so you can add a new migration called AddIdentitySchema using

```
dotnet ef migrations add AddIdentitySchema.
```

The final step is to update your application’s Razor Pages and layouts to reference the default identity UI. Normally, adding 30 new Razor Pages to your application would be a lot of work, but using the default Identity UI makes it a breeze.

14.4.3 Updating the Razor views to link to the Identity UI

Technically, you don’t *have* to update your Razor Pages to reference the pages included in the default UI, but you probably want to add the login widget to your app’s layout at a minimum. You’ll also want to make sure that your Identity Razor Pages use the same base Layout.cshtml as the rest of your application.

We’ll start by fixing the layout for your Identity pages. Create a file at the “magic” path Areas/Identity/Pages/_ViewStart.cshtml, and add the following contents:

```
@{ Layout = "/Pages/Shared/_Layout.cshtml"; }
```

This sets the default layout for your Identity pages to your application’s default layout. Next, add a _LoginPartial.cshtml file in Pages/Shared to define the login widget, as shown in the following listing. This is pretty much identical to the template generated by the default template, but using our custom `ApplicationUser` instead of the default `IdentityUser`.

Listing 14.7 Adding a _LoginPartial.cshtml to an existing app.

```
@using Microsoft.AspNetCore.Identity
@using RecipeApplication.Data;                             #A
@inject SignInManager<ApplicationUser> SignInManager       #B
@inject UserManager<ApplicationUser> UserManager           #B

<ul class="navbar-nav">
@if (SignInManager.IsSignedIn(User))
{
  <li class="nav-item">
    <a  class="nav-link text-dark" asp-area="Identity"
    asp-page="/Account/Manage/Index" title="Manage">Hello
    User.Identity.Name!</a>
  </li>
    <li class="nav-item">
      <form class="form-inline" asp-page="/Account/Logout"
      asp-route-returnUrl="@Url.Page("/", new { area = "" })"
      asp-area="Identity" method="post" >
        <button  class="nav-link btn btn-link text-dark"
          type="submit">Logout</button>
        </form>
    </li>
}
else
{
  <li class="nav-item">
    <a class="nav-link text-dark" asp-area="Identity"
      asp-page="/Account/Register">Register</a>
  </li>
  <li class="nav-item">
    <a class="nav-link text-dark" asp-area="Identity"
      asp-page="/Account/Login">Login</a>
  </li>
}
</ul>
```

#A Update to your project’s namespace that contains ApplicationUser
#B The default template uses IdentityUser. Update to use ApplicationUser instead.

This partial shows the current login status of the user and provides links to register or sign in. All that remains is to render the partial by calling

```
<partial name="_LoginPartial" />
```

in the main layout file of your app, _Layout.cshtml.

And there you have it: you’ve added Identity to an existing application. The Default UI makes doing this relatively simple, and you can be sure you haven’t introduced any security holes by building your own UI!

As I described in section 14.3.4, there are some features that the default UI doesn't provide that you need to implement yourself, such as email confirmation and 2FA QR code generation. It's also common to find that you want to update a single page, here and there. In the next section I show how you can replace a page in the default UI, without having to rebuild the entire UI yourself.

14.5 Customizing a page in ASP.NET Core Identity's default UI

In this section you'll learn how to use "scaffolding" to replace individual pages in the default Identity UI. You'll learn to scaffold a page so that it overrides the default UI, allowing you to customize both the Razor template and the `PageModel` page handlers.

Having Identity provide the whole UI for your application is great in theory, but in practice there are a few wrinkles, as you've already seen in section 14.3.4. The default UI provides as much as it can, but there's some things you may want to tweak. For example, both the login and register pages describe how to configure external login providers for your ASP.NET Core applications, as you saw in figures 14.12 and 14.13. That's useful information for you as a developer, but not something you want to be showing to your users! Another often-cited requirement is the desire to change the look and feel of one or more pages.

Luckily, the default Identity UI is designed to be incrementally replaceable, so that you can "override" a single page, without having to rebuild the entire UI yourself. On top of that, both Visual Studio and the .NET CLI have functions that allow you to "scaffold" any (or all) of the pages in the default UI, so that you don't have to start from scratch when you want to tweak a page.

> **DEFINITION** Scaffolding is the process of generating files in your project that serve as the basis for customization. The Identity scaffolder adds Razor Pages in the correct locations, so they override equivalent pages with the default UI. Initially, the code in the scaffolded pages matches that in the default Identity UI, but you are free to customize it.

As an example of the changes you can easily make, we'll scaffold the registration page, and remove the additional information section about external providers. The following steps describe how to scaffold the Register.cshtml page in Visual Studio. Alternatively, you can use the .NET CLI to scaffold the registration page.[65]

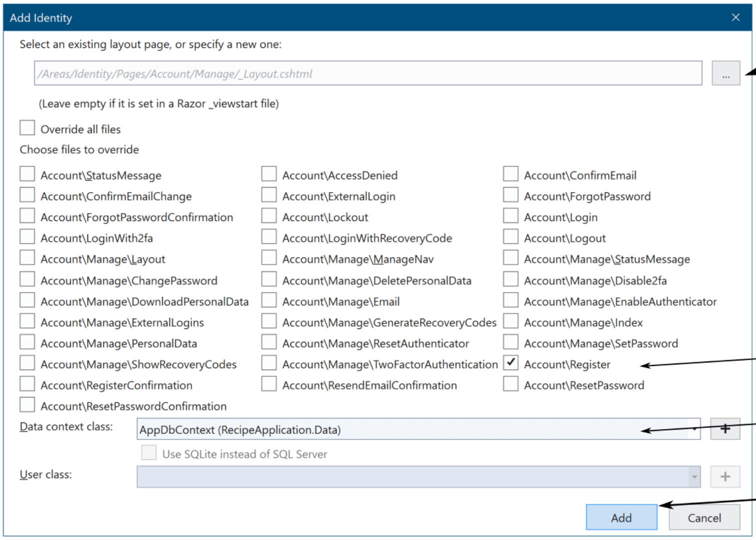
1. Add the Microsoft.VisualStudio.Web.CodeGeneration.Design and Microsoft.EntityFrameworkCore.Tools NuGet packages to your project file, if they're not

[65] Install the necessary .NET CLI tools and packages as described in the documentation https://docs.microsoft.com/aspnet/core/security/authentication/scaffold-identity. Then run `dotnet aspnet-codegenerator identity -dc RecipeApplication.Data.AppDbContext --files "Account.Register"`

already added. Visual Studio uses these packages to scaffold your application correctly, and without them you will get an error running the scaffolder.

```
<PackageReference Version="3.1.3"
    Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" />
<PackageReference Version="3.1.3"
    Include="Microsoft.EntityFrameworkCore.Tools" />
```

2. Ensure your project builds—if it doesn't build, the scaffolder will fail before adding your new pages.
3. Right-click your project and choose Add > New Scaffolded Item.
4. In the selection dialog, choose Identity from the category, and click Add.
5. In the Add Identity dialog, select the Account/Register page, and select your application's `AppDbContext` as the Data context class, as shown in figure 14.15. Click Add to scaffold the page.



Add Identity

Select an existing layout page, or specify a new one:

/Areas/Identity/Pages/Account/Manage/_Layout.cshtml

(Leave empty if it is set in a Razor _viewstart file)

□ Override all files

Choose files to override

□ Account\StatusMessage
□ Account\ConfirmEmailChange
□ Account\ForgotPasswordConfirmation
□ Account\LoginWith2fa
□ Account\Manage\Layout
□ Account\Manage\ChangePassword
□ Account\Manage\DownloadPersonalData
□ Account\Manage\ExternalLogins
□ Account\Manage\PersonalData
□ Account\Manage\ShowRecoveryCodes
□ Account\RegisterConfirmation
□ Account\ResetPasswordConfirmation
□ Account\AccessDenied
□ Account\ExternalLogin
□ Account\Lockout
□ Account\LoginWithRecoveryCode
□ Account\Manage\ManageNav
□ Account\Manage\DeletePersonalData
□ Account\Manage\Email
□ Account\Manage\GenerateRecoveryCodes
□ Account\Manage\ResetAuthenticator
□ Account\Manage\TwoFactorAuthentication
□ Account\ResendEmailConfirmation
□ Account\ConfirmEmail
□ Account\ForgotPassword
□ Account\Login
□ Account\Logout
□ Account\Manage\StatusMessage
□ Account\Manage\Disable2fa
□ Account\Manage\EnableAuthenticator
□ Account\Manage\Index
□ Account\Manage\SetPassword
☑ Account\Register
□ Account\ResetPassword

Data context class: AppDbContext (RecipeApplication.Data)

□ Use SQLite instead of SQL Server

User class:

Leave the layout field empty.

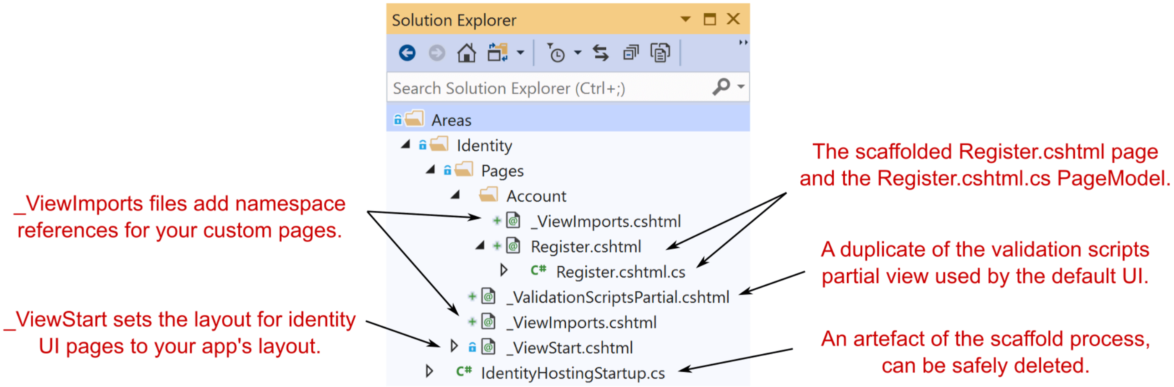
Override the Account\Register file.

Select the existing data context.

Click Add to scaffold the file.

Figure 14.15 Using Visual Studio to scaffold Identity pages. The generated Razor Pages will override the versions provided by the Default UI.

Visual Studio builds your application, and then generates the Register.cshtml page for you, placing it into the Areas/Identity/Pages/Account folder. It also generates several supporting files, as shown in figure 14.16. These are mostly required to ensure your new Register.cshtml Page can reference the remaining pages in the default Identity UI.



_ViewImports files add namespace references for your custom pages.

_ViewStart sets the layout for identity UI pages to your app's layout.

The scaffolded Register.cshtml page and the Register.cshtml.cs PageModel.

A duplicate of the validation scripts partial view used by the default UI.

An artefact of the scaffold process, can be safely deleted.

Figure 14.16 The Scaffolder generates the Register.cshtml Razor Page, along with supporting files required to integrate with the remainder of the default Identity UI.

We’re interested in the Register.cshtml page, as we want to customize the UI on the Register page, but if you look inside the code-behind page, Register.cshtml.cs, you’ll see how much complexity the default Identity UI is hiding from you. It’s not insurmountable (we’ll customize the page handler in section 14.6) but it’s always good to avoid writing code if you can help it!

Now you have the Razor template in your application, you can customize it to your heart’s content. The downside is that you’re now maintaining more code than you were with the default UI. You didn’t have to write it, but you may still have to *update* it when a new version of ASP.NET Core is released.

I like to use a bit of a “trick” when it comes to overriding default identity UI like this. In many cases, you don’t actually want to change the *page handlers* for the Razor Page, just the Razor *view*. You can achieve this by deleting the Register.cshtml.cs `PageModel` file, and pointing your newly-scaffolded cshtml file at the *original* `PageModel`, which is part of the default UI NuGet package.

The other benefit of this approach is that you can delete some of the other files that were auto-scaffolded. In total, you can make the following changes:

1. Update the `@model` directive in Register.cshtml to point to the default UI `PageModel`:

```
@model Microsoft.AspNetCore.Identity.UI.V4.Pages.Account.Internal.RegisterModel
```

2. Update Areas/Identity/Pages/_ViewImports.chstml to the following:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

3. Delete Areas/Identity/Pages/IdentityHostingStartup.cs
4. Delete Areas/Identity/Pages/_ValidationScriptsPartial.cshtml
5. Delete Areas/Identity/Pages/Account/Register.cshtml.cs
6. Delete Areas/Identity/Pages/Account/_ViewImports.cshtml

After making all these changes, you'll have the best of both worlds—you can update the default UI Razor Pages HTML, without taking on the responsibility of maintaining the default UI code behind.

> **TIP** In the source code for the book you can see these changes in action, where the Register view has been customized to remove the references to external identity providers.

Unfortunately, it's not always possible to use the default UI `PageModel`. Sometimes, you *need* to update the page handlers, such as when you want to change the functionality of your Identity area, rather than just the look and feel. A common requirement is needing to store additional information about a user, as you'll see in the next section.

14.6 Managing users: adding custom data to users

In this section you'll see how to customize the `ClaimsPrincipal` assigned to your users by adding additional claims to the AspNetUserClaims table when the user is created. You'll also see how to access these claims in your Razor Pages and templates.

Very often, the next step after adding Identity to an application is to customize it. The default templates only require an email and password to register. What if you need more details, like a friendly name for the user? Also, I've mentioned that we use claims for security, so what if you want to add a claim called `IsAdmin` to certain users?

You know that every user principal has a collection of claims so, conceptually, adding any claim just requires adding it to the user's collection. There are two main times that you would want to grant a claim to a user:

- *For every user, when they first register on the app.* For example, you might want to add a "Name" field to the Register form and add that as a claim to the user when they register.
- *Manually, after the user has already registered.* This is common for claims used as "permissions," where an existing user might want to add an `IsAdmin` claim to a specific user after they have registered on the app.

In this section, I show you the first approach, automatically adding new claims to a user when they're created. The latter approach is the more flexible and, ultimately, is the approach many apps will need, especially line-of-business apps. Luckily, there's nothing conceptually difficult to it; it requires a simple UI that lets you view users and add a claim through the same mechanism I'll show here.

> **TIP** Another common approach is to customize the `IdentityUser` entity, by adding a `Name` property for example. This approach is sometimes easier to work with if you want to give users the ability to edit that property. The documentation describes the steps required to achieve that: https://docs.microsoft.com/aspnet/core/security/authentication/add-user-data.